Basic Layouts

This section contains mixins and functions that you need to use for all Susy layouts.

Container (Mixin)

The `container()` mixin sets styles for the Susy container. It takes in a map as an argument and defaults to `$susy` if no arguments are given to it.

Susy will also add `background-image` properties to this container to display the background grid if `debug: image` is set to `show`.

```
// Scss
.wrap {
  @include container();
}
```

```
/* CSS */
.wrap {
  max-width: 100%;
  margin-left: auto;
  margin-right: auto;
  /* plus other background styles */
}
```

`$susy` settings that affect the container are:

```
// Scss
$susy: (
  container: auto (default) | length // Sets width of container
  container-position: center (default) | left | right | length // Sets left and right ma
  );
```

Span (Mixin)

The `span()` mixin creates 3 CSS properties: `float`, `width` and `margin (or padding)`. The type of margins produced depends on the `gutter-position` setting.

The easiest way to use the `span()` mixin is to use the following syntax:

```
// Scss
@include span($span of $context);
```

Where `$span` is the number of columns and `$context` is the total number of columns.

```
// Scss
.selector {
  @include span(3 of 4);
}
```

```
/* CSS */
.selector {
  float: left;
  margin-right: 5.26316%;
}
```

`$susy` settings that affect the container are:

```
$susy : (
  output : float (default) | isolate, // Determines whether to use float or isolate tech
  math: fluid (default) | static, // Determines units to use
  columns: number, // Determines total number of columns
  column-width: auto (default) | length, // Determines width of each column.
  gutters: number, // Determines gutter width relative to column-width
  gutter-position: after, // Determines the location of gutters. See below in gutter-pos
  );
```

Span (Function)

The `span()` function returns the width output from the `span()` mixin.

```
// Scss
.selector {
  width: span(3 of 4);
}
```

```
/* CSS */
.selector {
  width: 73.68421%;
}
```

Gutter (Function)

The `gutter()` function allows you to get the width of one gutter. It takes in the `$context` as an argument

```
// Scss
```

```
.one-gutter-of-a-twelve-column-grid {
  width: gutter(12);
}
```

```
/* CSS */
.one-gutter-of-a-twelve-column-grid {
  width: 1.69492%;
}
```

Gallery (Mixin)

The `gallery()` mixin creates a gallery with the isolate technique. It takes in the same arguments as the `span()` mixin.

```
// Scss
.gallery__item {
  @include gallery(4 of 16);
}
```

```
/* CSS */
.gallery__item {
  width: 24.05063%;
  float: left;
}

.gallery__item:nth-child(4n + 1) {
  margin-left: 0;
  margin-right: -100%;
  clear: both;
  margin-left: 0;
}

.gallery__item:nth-child(4n + 2) {
  margin-left: 25.31646%;
  margin-right: -100%;
  clear: none;
}

.gallery__item:nth-child(4n + 3) {
  margin-left: 50.63291%;
  margin-right: -100%;
  clear: none;
}
```

```css
.gallery__item:nth-child(4n + 4) {
  margin-left: 75.94937%;
  margin-right: -100%;
  clear: none;
}
```

Responsive Layouts

This section covers how to make Susy layouts responsive.

Media Queries

CSS media queries are required to create responsive layouts. It is recommended to create mobile-first layouts that uses `min-width` queries as much as possible.

`max-width` queries can also be used in addition to `min-width` queries to contain styles to a specific viewport.

Understanding media queries is extremely important. More information on this in Chapter 9 of the book.

Breakpoint (mixin)

The `breakpoint()` mixin helps to generate `min-width` and `max-width` media queries easily. It is made available to you if you add Breakpoint Sass to your project.

```
.min-width-query {
  @include breakpoint(300px) {
    color: red;
  }
}

.min-and-max-width-query {
  @include breakpoint(300px 600px) {
    color: blue;
  }
}

.max-width-query {
  @include breakpoint(max-width 600px) {
    color: green;
  }
}
```

```
/* CSS */
@media (min-width: 300px) {
  .min-width-query {
    color: red;
  }
}
```

```
@media (min-width: 300px) and (max-width: 600px) {
  .min-and-max-width-query {
    color: blue;
  }
}

@media (max-width: 600px) {
  .max-width-query {
    color: green;
  }
}
```

You'll have to use the `nested()` mixin or `with-layout()` mixin if you want to change the the column context at any specific breakpoint. More on that later.

```
.breakpoint {
  @include nested(6) {
    // Number of columns is set to 6 here
  }
}
```

Susy Breakpoint (Mixin)

The `susy-breakpoint()` mixin lets you add a breakpoint and a context in one mixin. Its syntax is:

```
// Scss
@include susy-breakpoint($query, $layout) {
  // Styles go here
}
```

Where `$query` refers to the media queries used and `$layout` refers to the new `$susy` map used in the breakpoint.

`susy-breakpoint()` uses the `breakpoint()` under the hood with the `with-layout()` mixin.

```
@include breakpoint($query) {
  @include with-layout($layout) {
    // Styles go here
  }
}
```

Note: If you are using asymmetric grids, use the `bp-with-context()` mixin mentioned below instead of `susy-breakpoint()`.

Note: You no longer need to add Breakpoint Sass to use the `susy-breakpoint()` mixin.

Show Grid (Mixin)

The `show-grid()` mixin creates the background grid, which lets you debug when working with responsive sites.

This `show-grid()` mixin is used on the container element.

```
.wrap {
  @include container();
  @include susy-breakpoint(300px, 8) {
    @include show-grid; // Shows 8-column grid
  }
}
```

Susy Contexts

Susy relies heavily on you to provided the correct `$context` to calculate the grid math. It looks for context in these 3 places:

1. The mixin that is used
2. `nested()` or `with-layout()` wrapper
3. `$susy` map

Susy will use the first `$context` it finds as the context.

```
// Scss
// Third search location
$susy: (
  columns: 4
);

.selector {
  // Second search location
  @include nested(8) {
    @include span(4 of 12); // First search location
  }
}
```

The output created in this example will be `@include span(4 of 12)`;

Nested (Mixin)

The `nested()` mixin changes the Susy context within the mixin to the number of columns given to it.

```
// Scss
.selector {
  @include nested(8) {
    @include span(4);
  }
}
```

This example is equivalent to `span(4 of 8)`;

With-layout (Mixin)

The `with-layout()` changes the whole `$susy` map within the mixin to the `$map` variable given to it.

```scss
// Scss
$new-map: (
  columns: 8,
  gutter-position: inside
);

@include with-layout($new-map) {
  .selector {
    @include span(4);
  }
}
```

`.selector` in this example will have a context of 8 columns and a `gutter-position` of `inside`.

Complex Susy Contexts

Contexts can get complicated with asymmetric layouts. It is recommend to use the following mixins from Susy helpers to get and set the context when working with asymmetric layouts.

With Context (Mixin)

The `with-context()` mixin gets a context from the `$contexts` map. Its syntax is:

```
// Scss
@include with-context($keys...) {
  // Styles go here
}
```

`$keys` refer to the map keys within the `$contexts` map. You can get a key nested deep within the `$contexts` map by using a comma to separate the keys. Here's an example:

```
// Scss
$contexts: (
  medium: 1 2 3,
  deep-context: (
    deep: 4 5 6
  )
);

// Medium Context (1 2 3)
@include with-context(medium) {
  // Styles go here
}

// Deep context (4 5 6) {
  @include with-context(deep-context, deep) {
  // Styles go here
  }
}
```

Bp With Context (Mixin)

The `bp-with-context()` mixin adds a breakpoint query to the `with-context()` mixin, just like how `susy-breakpoint()` adds a query to `breakpoint()`. It has the following syntax:

```
@include bp-with-context($query, $keys...) {
  // Styles go here
```

```
}
```

`bp-with-context()` requires the use of Breakpoint Sass (for now). The `$query` given to it is written in the same way as you would write a `breakpoint()` query.

Add Context (Mixin)

The `add-context()` mixin is a helper mixin to help add a context to the `$contexts` map. It has the following syntax:

```
@include add-context($context, $keys...);
```

`$context` refers to the context used and `$keys` refers to the same `$keys` used in `with-context()`.

It's recommended to use `with-context()` and `add-context()` together to store the correct context. Here's an example:

```
$contexts: (
  medium: 1 2 3 2 1;
)

@include with-context(medium) {
  @include span(2 at 2);
  @include add-context(2 at 2, medium-inner);
}

// Result
$contexts: (
  medium: 1 2 3 2 1,
  medium-inner: 2 3
)
```

Span Ac (Mixin)

The `span-ac()` mixin is a convenience mixin that combines the `span()` mixin with `add-context()` mixin. If we take the same example as above, you could have written this instead:

```
@include with-context(medium) {
  @include span-ac(2 at 2, medium-inner);
}
```

Multiple Grids With Susy

You can use the `with-layout()` mixin to help create multiple Susy maps for different parts of the site:

```
.layout1 {
  @include with-layout($map1) {
    // Styles go here
  }
}

.layout2 {
  @include with-layout($map2) {
    // Styles go here
  }
}
```

The Susy Shorthand

The Susy shorthand allows you to quickly overwrite Susy global settings with local settings of your choice. It works with any Susy mixin or function, but is mostly used with the `span()` mixin.

It has the following syntax:

```
// Scss
$shorthand: $span of $grid $location $keywords
```

Span

`$span` refers to the width of the element you're creating. It is usually given a unit-less number.

```
.selector {
  @include span(3); // This means 3 columns
}
```

Grid

`$grid` refers to the columns and gutters of the grid you're creating. It's a combination of `columns`, `gutters` and `gutter-width` settings.

Here are the possible combinations:

```
// Scss
$grid: 12 // columns
$grid: 12 (1/3) // 12 columns with 1/3 gutters settings
$grid: 12 (60px 10px) // 12 columns, 60px column-width and 10px gutters (or 1/6 gutters)
```

It is always preceded by an `of` keyword.

```
// Scss
.selector {
  @include span(3 of 12 (60px 10px));
}
```

Location

`$location` refers to the place where the element is supposed to be placed at. These `$locations` can be either `first`, `last` or `at <number>`.

`first` tells Susy to output the element at the first column.
`last` tells Susy to output the element at the last column.
`at 3` tells Susy to output the element at the 3rd column.

Keywords

There are two types of keywords you can use with Susy mixins and functions – Global and Local keywords.

Global keywords are keywords for settings within the `$susy` map. They are explained in detail in the chapter on Susy Settings.

Local keywords on the other hand, are keywords that can only be used with mixins and functions. There are 4 local keywords and they can take the following values:
spread: `narrow (default) | wide | wider`
role: `null (default) | nest`
clear: `null (default) | break | nobreak`
gutter-override: `null (default) | no-gutters | no-gutter`
You can find more information about the Susy shorthand in the Shorthand chapter.

Susy Grid Types

There are 3 common types of grids people will use when working with Susy – Fluid, Static and Asymmetric grids. This section will provide you with the minimum number of settings in the `$susy` map to create this kind of grid.

Fluid

Nothing is needed. Fluid grids are the default.

Static

```
// Scss
$susy: (
  math: static,
  column-width: 90px,
  container: auto
  );
```

Asymmetric

```
$susy: (
  output: isolate,
  columns: 1 2,
  );
```

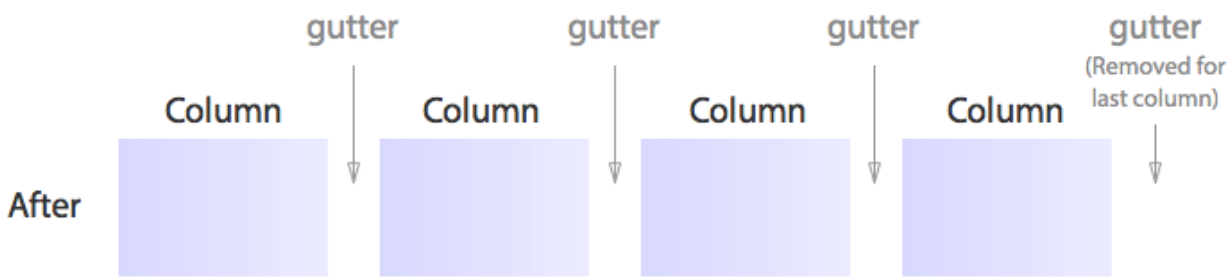
Gutter Positions

The `gutter-position` setting changes the location where gutters are created. It also determines whether gutters are created as margins or paddings.

After

The `after` position is the default `gutter-position` setting. Susy adds gutters as margins after each column when using the `after` position.

```
$susy: ( gutter-position: after );
```

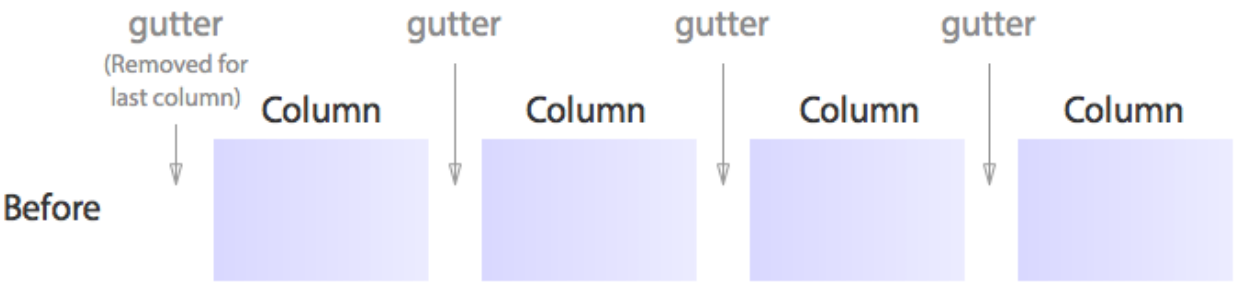


You need to remove the gutter on the final item in the row by either setting `@include last()` or `margin-right: 0` on it.

Before

Susy adds gutters as margins before each column when using the `after` position.

```
$susy: ( gutter-position: before );
```

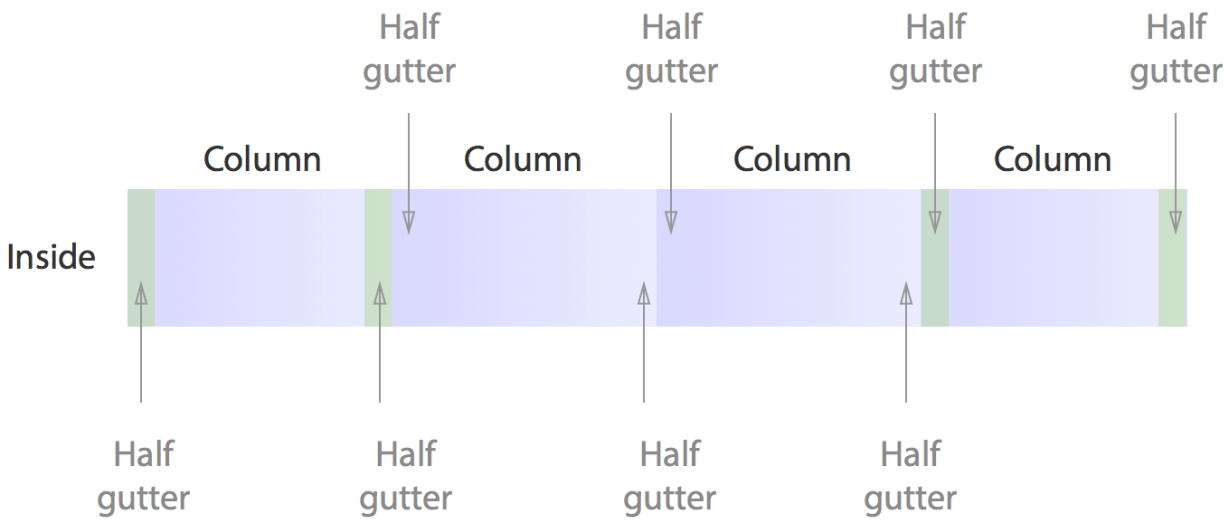


You will need to remove the gutter of the first item either by using `@include last()` or by setting `margin-left: 0`.

Split

Susy splits gutters up into two and places them as margins at both sides of every column if the gutter position is set to `split`.

```
$susy: ( gutter-position: split );
```



You will need to add a `nest` keyword to the parent element of the nested element if the parent is spanned.

There is no need to remove gutters from the extreme edges of the grids.

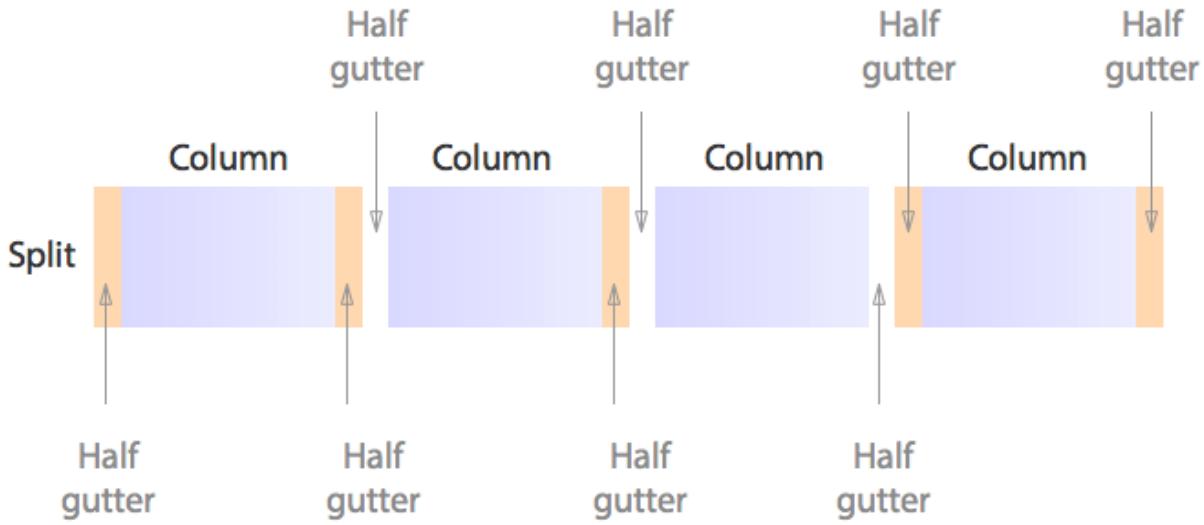
Inside

Susy splits up gutters into two and adds them as paddings to both sides of each column.

```
$susy: ( gutter-position: inside );
```

Similar to `split`, you will need to add a `nest` keyword to the parent element of the nested element if the parent is spanned.

There is no need to remove gutters from the extreme edges of the grids.



Inside-static

Inside-static is similar to `inside`. The difference is that Susy adds the paddings with a unit instead of a percentage.

In this case, a `column-width` declaration is needed as well.

```
$susy: (
  column-width: 60px,
  gutter-position: inside
  );
```

Isolate Technique

When using the Isolate Technique to layout Susy items, be sure to add the `$location` keyword.

```
.selector {
  @include span(3 at 2);
}
```