Susy Documentation

Release 2.2.12

Miriam Eric Suzanne and contributors

April 10, 2018

Contents

In a world of agile development and super-tablet-multi-magic-laptop-phones, the best layouts can't be contained in a single framework or technique. CSS Libraries are a bloated mess of opinions about how to do your job. Why let the table-saw tell you where to put the kitchen?

Your markup, your design, your opinions | *our math*.

Note: These docs are focused on Susy 2.2.12. See the *Susy One* documentation for help with earlier versions of Susy.

CHAPTER 1

Contents

Getting Started

The only requirement is Sass, but Susy was built to be part of the Compass ecosystem, and we recommend pairing with tools like Breakpoint and Vertical Rhythms.

Compass is still required for the *Susy One* syntax.

Simple Install

```
# command line
gem install susy
```

Bundler or Rails

Warning: In order to use Susy 2 with Rails you must update your Gemfile to use sass-rails ~> 5.0.0. This is because Susy 2 requires Sass >= 3.3 whilst Rails 4.1 and below include a version of sass-rails which does not support Sass 3.3.

```
# Gemfile
# gem 'sass-rails', '~> 4.0.3'
gem 'sass-rails', '~> 5.0.0'
gem 'susy'

# If you want Compass:
gem 'compass-rails', '~> 2.0.0'

# config/application.rb
require 'susy'

# command line
bundle install
```

If you add Susy to an existing Rails app, follow the steps above, but use bundle update instead of bundle install.

```
# command line
bundle update
```

Webpack and npm

Install using npm:

```
npm install susy sass-loader --save-dev
```

Make sure you have sass-loader enabled in your *webpack* configuration:

```
// webpack.config.js
loaders: [
  {
    test: /\.scss$/,
    loader: 'style!css!sass'
  }
]
```

Start using Susy:

```
/* app.scss */
@import "~susy/sass/susy";
```

Gulp

Install susy with npm:

```
npm install susy --save-dev
```

Add Gulp Task:

```
// gulpfile.js
gulp.task('sass', function() {
  return gulp.src('scss/*.scss')
      .pipe(sass({
          outputStyle: 'compressed',
          includePaths: ['node_modules/susy/sass']
      }).on('error', sass.logError))
      .pipe(gulp.dest('dist/css'));
});
```

Start using Susy:

```
/* app.scss */
@import "susy";
```

Grunt (and Yeoman)

You can enable Susy in Grunt by adding a line to your `Gruntfile.js`. You will need to add a line to either your Sass task or, if you're using Compass, your Compass task.

To add Susy to the Sass task, edit your Gruntfile.js at the root level of your project and look for the Sass-related rules. Add `require: 'susy'` inside the `options` object:

```
// Gruntfile.js
sass: {
  dist: {
    options: {
      style: 'expanded',
```

```
      require: 'susy'
    },
    files: {
        'css/style.css': 'scss/style.scss'
    }
  }
}
```

Assuming you've already installed Susy, it will now be added to the project and will not clash with Yeomans grunt rules.

To add Susy to the Compass task, edit your Gruntfile.js at the root level of your project and look for the Compass-related rules. Add `require: 'susy'` inside the `options` object:

```
// Gruntfile.js
compass: {
    options: {
      require: 'susy',
      ...
    }
  }
}
```

Again, assuming you've already installed Susy, it will now be added to the project.

Bower

```
# command line
bower install susy --save
```

This will add the Susy repository to your `bower_components` directory or create a `bower_components` directory for you.

```
// Import Susy
@import "bower_components/susy/sass/susy";
```

You can also import Susyone.

```
// Import Susy
@import "bower_components/susy/sass/susyone";
```

Compass

If you want to use Susy with Compass, start by installing Compass.

Create a new Compass project:

```
# command line
compass create --using susy <project name>
```

Alternatively, add Susy to a current project

```
# command line
compass install susy
```

Manual Start

If you want to copy in the Sass files directly, and skip any package management, you can do that too.

- Download the zip file from GitHub.
- Copy the contents of the "sass" folder (feel free to remove everything else).
- Paste the files in your project "sass" folder (whatever you call it).

Version Management

When you work with bundled gems across a number of different projects, managing gem versions can become an issue.

If you are in a Ruby environment, check out RVM. In a Python environment, we recommend virtualenv with these scripts added to your "postactivate" and "predeactivate" files.

Once you have that in place, Bundler can be used in either environment to manage the actual installation and updating of the gems.

Quick Start

Once you have everything installed, you can import Susy into your Sass files.

```
@import "susy";
```

The basic Susy layout is composed using two simple mixins:

```
@include container; // establish a layout context
@include span(<width>); // lay out your elements
```

For example:

```
body { @include container(80em); }
nav { @include span(25%); }
```

If you want to lay your elements out on a grid, you can use the `span` mixin to calculate column widths:

```
nav { @include span(3 of 12); }
```

But you don't have to do things the Susy way. We give you direct access to the math, so you can use it any way you like:

```
main {
  float: left;
  width: span(4);
  margin-left: span(2) + gutter();
  margin-right: gutter();
}
```

You can also establish *global settings*, to configure Susy for your specific needs. Create a `$susy` variable, and add your settings as a map.

```
$susy: (
  columns: 12,  // The number of columns in your grid
  gutters: 1/4, // The size of a gutter in relation to a single column
);
```

There are many more settings available for customizing every aspect of your layout, but this is just a quick-start guide. Keep going to get the details.

Settings

The new syntax for Susy is based around a number of settings that can be written either as a Sass Map or using a *shorthand syntax*. These two definitions are interchangeable:

```
$susy: (
  columns: 12,
  gutters: 1/4,
  math: fluid,
  output: float,
  gutter-position: inside,
);

$shorthand: 12 1/4 fluid float inside;
```

Either format can be passed as a single argument to the functions and mixins that make up the Susy language. Maps can even be used as part of the shorthand:

```
$susy: (
  columns: 12,
  gutters: .25,
  math: fluid,
);

@include layout($susy float inside);
```

Unless otherwise noted, most settings can be controlled both globally (by setting the site-wide default) or locally (passed to individual functions and mixins).

Global Defaults

Here are all the global Susy settings with their default values:

```
$susy: (
  flow: ltr,
  math: fluid,
  output: float,
  gutter-position: after,
  container: auto,
  container-position: center,
  columns: 4,
  gutters: .25,
  column-width: false,
  global-box-sizing: content-box,
  last-flow: to,
  debug: (
    image: hide,
    color: rgba(#66f, .25),
    output: background,
    toggle: top right,
  ),
```

```
  use-custom: (
    background-image: true,
    background-options: false,
    box-sizing: true,
    clearfix: false,
    rem: true,
  )
);
```

You can set your own global defaults, or create individual layout maps to access as needed.

For global settings, create a `$susy` variable with any values that you need.

```
$susy: (
  columns: 12,
  gutters: .25,
  gutter-position: inside,
)
```

Layout

A "layout" in Susy is made up of any combination of settings. Layouts are stored as maps, but can also be written as *shorthand*.

Layout [function]

Convert *shorthand* into a map of settings.

`function`

> **Format** `layout(`*`$layout`*`)`
>
> **$layout** *<layout>*

```
// function input
$map: layout(auto 12 .25 inside fluid isolate);

//output
$map: (
  container: auto,
  columns: 12,
  gutters: .25,
  gutter-position: inside,
  math: fluid,
  output: isolate,
);
```

This is useful any time you need to combine settings stored in different variables. It's not possible to combine two shorthand variables:

```
// THIS WON'T WORK
$short: 12 .25 inside;
@include layout($short fluid no-gutters);
```

but it is possible to add a map into the shorthand:

```
// THIS WILL WORK
$map: layout(12 .25 inside);
@include layout($map fluid no-gutters);
```

or combine two maps:

```
$map1: 13 static;
$map2: (6em 1em) inside;
@include layout($map1 $map2);
```

Layout [mixin]

Set a new layout as the global default.

mixin

> **Format** `layout($layout, $clean)`
>
> **$layout** *<layout>*
>
> **$clean** <boolean>

```
// mixin: set a global layout
@include layout(12 1/4 inside-static);
```

By default, these new settings are added to your existing global settings. Use the *$clean* argument to establish new settings from scratch.

With Layout

Temporarily set defaults for a section of your code.

mixin

> **Format** `with-layout($layout, $clean) { @content }`
>
> **$layout** *<layout>*
>
> **$clean** <boolean>
>
> **@content** Sass content block

```
@include with-layout(8 static) {
  // Temporary 8-column static grid...
}

// Global settings are restored...
```

By default, these new settings are added to your existing global settings. Use the *$clean* argument to establish new settings from scratch.

Susy-Get

function

> **Format** `susy-get($key, $layout)`
>
> **$key** Setting name
>
> **$layout** *<layout>*

You can access your layout settings at any time, using the `susy-get` function.

```
$large: layout(80em 24 1/4 inside);
$large-container: susy-get(container, $large);
```

To access a nested setting like `debug/image`, send the full path as a list for the `$key` argument.

```
$debug-image: susy-get(debug image);
```

If no setting is available (or no `$layout` is provided) `susy-get` falls back to the global user settings, and finally to the Susy default settings.

Flow

The reading direction of your document. Layout elements will stack out in the direction of flow, unless otherwise directed.

setting

> **Key** `flow`
>
> **Scope** global, local
>
> **Options** `rtl|ltr`
>
> **Default** `ltr`

ltr Layout elements will flow from left to right.

rtl Layout elements will flow from right to left.

Math

Susy can produce either relative widths (fluid percentages) or static widths (using given units).

setting

> **Key** `math`
>
> **Scope** global, local
>
> **Options** `fluid|static`
>
> **Default** `fluid`

fluid All internal grid spans will be calculated relative to the container, and output as `%` values.

static All internal grid values will be calculated as multiples of the `column-width` setting. If you set column-width to `4em`, your grid widths will be output as `em` values.

Output

Susy can generate output using different layout techniques. Currently we have a float module, with an extension to handle isolation as well. In the future there could be flexbox, grid, and other output styles.

setting

Key `output`

Scope global, local

Options `float|isolate`

Default `float`

float Floats are the most common form of layout used on the web.

isolate Isolation is a trick developed by John Albin Wilkins to help fix sub-pixel rounding bugs in fluid, floated layouts. You can think of it like absolute positioning of floats. We find it to be very useful for spot-checking the worst rounding bugs, but we think it's overkill as a layout technique all to itself.

Container

Set the max-width of the containing element.

setting

Key container

Scope global, local [container only]

Options `<length>|auto`

Default `auto`

<length> Set any explicit length (e.g. `60em` or `80%`), and it will be applied directly to the container.

auto Susy will calculate the width of your container based on the other grid settings, or fall back to `100%`.

> **Warning:** For `static` layouts, leave `container: auto` and set the `column-width` instead. Susy will calculate the outer container width for you. Dividing columns out of a set container width would leave you open to sub-pixel errors, and no one likes sub-pixel errors.

Container Position

Align the container relative to it's parent element (often the viewport).

setting

Key container-position

Scope global, local [container only]

Options `left|center|right|<length> [*2]`

Default `center`

left Holds container elements flush left, with `margin-left: 0;` and `margin-right: auto;`.

center Centers the container, by setting both left and right margins to `auto`.

right Pushes the container flush right, with `margin-right: 0;` and `margin-left: auto;`.

<length> [*2] If one length is given, it will be applied to both side margins, to offset the container from the edges of the viewport. If two values are given, they will be used as `left` and `right` margins respectively.

Columns

Establish the column-count and arrangement for a grid.

setting

Key `columns`

Scope global, local

Options `<number>|<list>`

Default `4`

<number> The number of columns in your layout.

<list> For asymmetrical grids, list the size of each column relative to the other columns, where `1` is a single column-unit. `(1 2)` would create a 2-column grid, with the second column being twice the width of the first. For a Fibonacci-inspired grid, use `(1 1 2 3 5 8 13)`.

Gutters

Set the width of a gutter relative to columns on your grid.

setting

Key `gutters`

Scope global, local

Options `<ratio>`

Default `1/4`

<ratio> Gutters are established as a ratio to the size of a column. The default `1/4` setting will create gutters one quarter the size of a column. In asymmetrical grids, this is `1/4` the size of a single column-unit.

If you want to set explicit column and gutter widths, write your `gutters` setting as `<gutter-width>/<column-width>`. You can even leave the units attached.

```
// 70px columns, 20px gutters
$susy: (
  gutters: 20px/70px,
);
```

Column Width

Optionally set the explicit width of a column.

setting

Key `column-width`

Scope global, local

Options `<length>|false/null`

Default `false`

<length> The width of one column, using any valid unit. This will be used in `static` layouts to calculate all grid widths, but can also be used by `fluid` layouts to calculate an outer maximum width for the container.

false/null There is no need for column-width in `fluid` layouts unless you specifically want the container-width calculated for you.

Gutter Position

Set how and where gutters are added to the layout, either as padding or margins on layout elements.

setting

Key gutter-position

Scope global, local

Options `before|after|split|inside|inside-static`

Default `after`

before Gutters are added as `margin` before a layout element, relative to the flow direction (left-margin for ltr, right-margin for rtl). The first gutter on each row will need to be removed.

after Gutters are added as `margin` after a layout element, relative to the flow direction. The last gutter on each row will need to be removed.

split Gutters are added as `margin` on both sides of a layout element, and are not removed at the edges of the grid.

inside Gutters are added as `padding` on both sides of a layout element, and are not removed at the edges of the grid.

inside-static Gutters are added as static `padding` on both sides of a layout element, even in a fluid layout context, and are not removed at the edges of the grid.

Global Box Sizing

Tell Susy what box model is being applied globally.

setting

Key `global-box-sizing`

Scope global

Options `border-box|content-box`

Default `content-box`

`content-box` Browsers use the `content-box` model unless you specify otherwise.

`border-box` If you are using the Paul Irish universal border-box technique (or something similar), you should change this setting to `border-box`. You can also use our `border-box-sizing` mixin, and we'll take care of it all for you.

For more, see the MDN box-sizing documentation.

Last Flow

The float-direction for the last element in a row when using the *float* output.

`setting`

Key `last-flow`

Scope global

Options `from|to`

Default `to`

`from` This is the default for all other elements in a layout. In an `ltr` (left-to-right) flow, the from-direction is `left`, and this setting would float "last" elements to the left, along with the other elements.

`to` In many cases (especially with `fluid` grids), it can be helpful to float the last element in a row in the opposite direction.

Debug

Susy has a few tools to help in debugging your layout as you work. These settings help you control the debugging environment.

`setting block`

Key `debug`

Scope global, local [container only]

Options <map of sub-settings>

```
$susy: (
  debug: (
    image: show,
    color: blue,
    output: overlay,
    toggle: top right,
  ),
);
```

Warning: Grid images are not exact. Browsers have extra trouble with sub-pixel rounding on background images. These are meant for rough debugging, not for pixel-perfect measurements. Expect the `to` side of your grid image (`right` if your flow is `ltr`) to be off by several pixels.

Debug Image

Toggle the available grid images on and off.

setting

Key `debug image`

Scope global, local [container only]

Options `show|hide|show-columns|show-baseline`

Default `hide`

show Show grid images, usually on the background of container elements, for the purpose of debugging. If you are using Compass vertical rhythms (or have set your own `$base-line-height` variable) Susy will show baseline grids as well.

hide Hide all grid debugging images.

show-columns Show only horizontal grid-columns, even if a baseline grid is available.

show-baseline Show only the baseline grid, if the `$base-line-height` variable is available.

Debug Output

The debug image can be output either as a background on the container, or as a generated overlay.

setting

Key `debug output`

Scope global, local [container only]

Options `background|overlay`

Default `background`

background Debugging images will be generated on on the background of the container element.

overlay Debugging images will be generated as an overlay using the container's `::before` element. Initially, the overlay is hidden, until you hover over the *toggle* we place in a corner of the viewport. Hover over the toggle to make the overlay appear.

Debug Toggle

If you are using the grid overlay option, Susy will generate a toggle to show/hide the overlay. Hovering over the toggle will show the overlay. You can place the toggle in any corner of the viewport using a combination of `top`, `right`, `bottom`, and `left`.

setting

Key `debug toggle`

Scope global

Options `right|left` and `top|bottom`

Default `top right`

Debug Color

Change the color of columns in the generated grid image.

setting

Key `debug color`

Scope global

Options <color>

Default `rgba(#66f, .25)`

Custom Support

There are several common helpers that you can tell Susy to use, if you provide them yourself or through a third-party library like Compass or Bourbon.

Custom Clearfix

Tell Susy to use a global `clearfix` mixin.

setting

Key `use-custom clearfix`

Scope global

Options <boolean>

Default `false`

false Susy will use an internal micro-clearfix.

true Susy will look for an existing `clearfix` mixin, and fallback to the internal micro-clearfix if none is found.

Custom Background Image

Tell Susy to use a global `background-image` mixin. This is only used for debugging.

setting

Key `use-custom background-image`

Scope global

Options <boolean>

Default `true`

false Susy will output background-images directly to CSS.

true Susy will look for an existing `background-image` mixin (like the ones provided by Compass and Bourbon), and fallback to plain CSS output if none is found.

Custom Background Options

Tell Susy to use global `background-size`, `-origin`, and `-clip` mixins. This is only used for debugging.

setting

Key `use-custom background-options`

Scope global

Options <boolean>

Default `false`

false Susy will output background-options directly to CSS.

true Susy will look for existing `background-size`, `-origin`, and `-clip` mixins (like the ones provided by Compass and Bourbon), and fallback to plain CSS output if none is found.

Custom Breakpoint Options

Tell Susy to use a custom `breakpoint` mixin, like the one provided by the Breakpoint plugin.

setting

Key `use-custom breakpoint`

Scope global

Options <boolean>

Default `true`

false Susy will use an internal fallback for media-queries.

true Susy will look for existing an `breakpoint` mixin like the one provided by the [Breakpoint](http://breakpoint-sass.com) plugin, and fallback to internal media-query support if none is found.

Custom Box Sizing

Tell Susy to use a global `box-sizing` mixin.

setting

Key `use-custom box-sizing`

Scope global

Options <boolean>

Default `true`

false Susy will output `box-sizing` official syntax, as well as `-moz` and `-webkit` prefixed versions.

true Susy will look for an existing `box-sizing` mixin (like the ones provided by Compass and Bourbon), and fallback to mozilla, webkit, and official syntax if none is found.

Custom Rem

Tell Susy to use the Compass `rem` support module.

`setting`

Key `use-custom rem`

Scope global

Options <boolean>

Default `true`

`false` Susy will output length values directly to CSS.

`true` Susy will look for an existing `rem` mixin, and check the `$rhythm-unit` and `$rem-with-px-fallback` settings provided by Compass, or fallback to plain CSS output if they aren't found.

Location

Reference a specific column on the grid for *row edges*, *isolation*, or *asymmetrical layouts*. Locations keywords don't require the `at` flag.

`setting`

Key `location`

Scope local

Options `first/alpha | last/omega | <number>`

Default `null`

`first, alpha` Set location to `1`.

`last, omega` Set the location to the final column, and any previous columns included by the relevant `span`.

`<number>` Set the location to any column-index between `1` and the total number of available columns.

Box Sizing

Set a new box model on any given element.

`setting`

Key `box-sizing`

Scope local

Options `border-box | content-box`

Default `null`

`border-box` Output `box-sizing` CSS to set the `border-box` model.

`content-box` Output `box-sizing` CSS to set the `content-box` model.

Spread

Adjust how many gutters are included in a column span.

setting

Key `spread`

Scope local

Options `narrow|wide|wider`

Default various...

narrow In most cases, column-spans include the gutters *between* columns. A span of `3 narrow` covers the width of 3 columns, as well as 2 internal gutters. This is the default in most cases.

wide Sometimes you need to include one side gutter in a span width. A span of `3 wide` covers the width of 3 columns, and 3 gutters (2 internal, and 1 side). This is the default for several margin/padding mixins.

wider Sometimes you need to include both side gutters in a span width. A span of `3 wider` covers the width of 3 columns, and 4 gutters (2 internal, and 2 sides).

Gutter Override

Set an explicit one-off gutter-width on an element, or remove its gutters entirely.

setting

Key `gutter-override`

Scope local

Options `no-gutters/no-gutter|<length>`

Default `null`

no-gutters, no-gutter Remove all gutter output.

<length> Override the calculated gutter output with an explicit width.

Role

Mark a grid element as a nesting context for child elements.

setting

Key `role`

Scope local

Options `nest`

Default `null`

nest Mark an internal grid element as a context for nested grids.

Note: This can be used with any grid type, but it is required for nesting with `split`, `inside`, or `inside-static` gutters.

Shorthand

Susy provides a shorthand syntax to easily pass arbitrary settings into functions and mixins. This allows the syntax to stay simple and readable for the majority of use cases, and only add complexity if/when you really need it.

```
// Establish an 80em container
@include container(80em);

// Span 3 of 12 columns
@include span(3 of 12);

// Setup the 960 Grid System
@include layout(12 (60px 20px) split static);

// Span 3 isolated columns in a complex, asymmetrical grid, without gutters
@include span(isolate 3 at 2 of (1 2 3 4 3 2 1) no-gutters);
```

Overview

In most cases, the syntax order is not important, but there are a few rules to get you started. The syntax generally breaks down into three parts.

syntax

Shorthand `$span $grid $keywords;`

span A `span` can be any length, or (in some cases) a unitless number representing columns to be spanned. The specifics change depending on the function or mixin where it is being passed. Some mixins even allow multiple spans, using the standard CSS TRBL <TOP RIGHT BOTTOM LEFT> syntax.

grid The `grid` is composed of a *Columns* setting, and optional settings for *Gutters* and *Column Width*. It looks something like this:

```
// 12-column grid
$grid: 12;

// 12-column grid with 1/3 gutter ratio
$grid: 12 1/3;

// 12-column grid with 60px columns and 10px gutters
$grid: 12 (60px 10px);

// asymmetrical grid with .25 gutter ratio
$grid: (1 2 3 2 1) .25;
```

keywords The `keywords` are the easiest. Most *settings* have simple keyword values that can be included in any order — before, after, or between the *span* and *grid* options.

```
// All the keywords in Susy:

$global-keywords: (
  container            : auto,
  math                 : static fluid,
  output               : isolate float,
  container-position   : left center right,
  flow                 : ltr rtl,
```

```
  gutter-position      : before after split inside inside-static,
  debug: (
    image              : show hide show-columns show-baseline,
    output             : background overlay,
  ),
);

$local-keywords: (
  box-sizing           : border-box content-box,
  edge                 : first alpha last omega,
  spread               : narrow wide wider,
  gutter-override      : no-gutters no-gutter,
  clear                : break nobreak,
  role                 : nest,
);
```

The global keywords can be used anywhere, and apply to global default *settings*. The local keywords are specific to each individual use.

Layout

The simplest shorthand variation is used for defining your layout in broad terms.

`shorthand`

> **Pattern** `<grid> <keywords>`

Nothing here is required — all the settings are optional and have global defaults. *grid* and *keyword* settings work exactly as advertised.

```
// grid: (columns: 4, gutters: 1/4, column-width: 4em);
// keywords: (math: fluid, gutter-position: inside-static, flow: rtl);
$small: 4 (4em 1em) fluid inside-static rtl;
```

You can easily convert layouts from shorthand to map syntax using the *Layout* function.

Span

Most of Susy's functions & mixins are used to calculate or set a width, or `span`.

`shorthand`

> **Pattern** `<span> at <location> of <layout>`

Most spans in Susy are either a unitless number (representing columns) or an explicit width. Some of them also require a location (particularly for asymmetrical grids and isolation).

The standard span syntax looks like this:

```
// Pattern:
$span: $span at $location of $layout;

// span: 3;
// location: 4;
// layout: (columns: 12, gutters: .25, math: fluid)
```

```
$span: 3 at 4 of 12 .25 fluid;

// Only $span is required in most cases
$span: 30%;
```

The "at" flag comes immediately before the location (unless the location itself is a keyword), and everything after the "of" flag is treated as part of the layout.

Some mixins accept multiple spans, using the common CSS "top right bottom left" (TRBL) pattern, or have other specific options. Those are all documented as part of the function/mixin details.

Toolkit

The Susy 2.0 toolkit is built around our *shorthand syntax*. Use the shorthand to control every detail, and adjust your defaults on-the-fly, so you are never tied down to just one grid, or just one output style.

Span [mixin]

Set any element to span a portion of your layout. For a floated or isolated layout, this will add necessary floats, widths, and margins.

mixin

> **Format** `span($span) { @content }`
>
> **$span** **
>
> **@content** Sass content block

There are many ways to use the span mixin...

Arbitrary Widths

For the simplest use, pass any width directly to the mixin:

```
// arbitrary width
.item { @include span(25%); }

// float output (without gutters)
.item {
  float: left;
  width: 25%;
}
```

Grid Widths

If you are using a grid, you can also span columns on the grid:

```
// grid span
.item { @include span(3); }

// output (7-column grid with 1/2 gutters after)
.item {
```

```
  float: left;
  width: 40%;
  margin-right: 5%;
}
```

Row Edges

When you use a grid with gutters `before` or `after`, you sometimes need to mark the `first` or `last` elements in a row, so Susy can remove the extra gutters:

```
// grid span
@include span(last 3);

// output (same 7-column grid)
.item {
  float: right;
  width: 40%;
  margin-right: 0;
}
```

For legacy reasons, `alpha` and `omega` can be used in place of `first` and `last`.

Context

Context is required any time you are using fluid math, and nesting grid elements inside other elements:

```
// 10-column grid
.outer {
  @include span(5);
  .inner { @include span(2 of 5); }
}
```

The `of` flag is used to signal context. The context is always equal to the grid-span of the parent. In some cases, you can imply changes in context by nesting elements inside the span tag itself:

```
// 10-column grid
.outer {
  // out here, the context is 10
  @include span(5) {
    // in here, the context is 5
    .inner { @include span(2); }
  }
}
```

Nesting

Grids with `inside`, `inside-static`, or `split` gutters don't need to worry about the edge cases, but they do have to worry about nesting.

If an element will have grid-aligned children, you should mark it as a `nest`:

```
// inside, inside-static, or split gutters
.outer {
  @include span(5 nest);
  .inner { @include span(2 of 5); }
}
```

Location

Asymmetrical grids and isolated output also need to know the desired `location` of the span. In both cases, use the `at` flag to set a location.

For isolation, you can use either an arbitrary width or a column index (starting with 1). For asymmetrical grid spans, the location setting must be a column index:

```
.width { @include span(isolate 500px at 25%); }
.index { @include span(isolate 3 at 2); }
```

narrow, wide, and wider

By default, a grid span only spans the gutters *between* columns. So a span of `2` includes 1 internal gutter (`narrow`). In some cases you want to span additional gutters on either side. So that same span of 2 could include the internal gutter, and one (`wide`) or both (`wider`) external gutters.

```
// grid span
.narrow { @include span(2); }
.wide { @include span(2 wide); }
.wider { @include span(2 wider); }

// width output (7 columns, .25 gutters)
// (each column is 10%, and each gutter adds 2.5%)
.narrow { width: 22.5%; }
.wide { width: 25%; }
.wider { width: 27.5%; }
```

If you are using inside gutters, the spans are wide by default but can be overridden manually.

Other Settings

Use the `full` keyword to span the entire context available, use `break` to start a new *Rows & Edges* by clearing previous floats, and `nobreak` to clear none. Use `no-gutters` to remove gutter output from an individual span, and use `border-box` or `content-box` to output changes in *box-sizing* on the fly.

You can set an arbitrary gutter override, by passing a map (e.g. `(gutter-override: 1.5em)`) as part of the shorthand syntax.

You can also change the *output* style, grid context, and other *global settings* on the fly:

```
// grid span
.item { @include span(isolate 4 at 2 of 8 (4em 1em) inside rtl break); }

// output
.item {
  clear: both;
  float: right;
  width: 50%;
  padding-left: .5em;
  padding-right: .5em;
  margin-left: 25%;
  margin-right: -100%;
}
```

Span [function]

The span function is identical to the *span mixin*, but returns only the span width value, so you can use it with custom output.

function

> **Format** `span($span)`
>
> **$span** **

```
.item {
  width: span(2);
  margin-left: span(3 wide);
  margin-right: span(1) + 25%;
}
```

Gutters

function/mixin

> **Format** `gutters($span)`
>
> **Alternate** `gutter($span)`
>
> **$span** **

Use `gutter` or `gutters` as a **function** to return the width of a gutter given your settings and current context.

```
// default context
margin-left: gutter();

// nested in a 10-column context
margin-left: gutter(10);
```

Use the **mixin** version to apply gutters to any element. Gutters are output as `margin` or `padding` depending on the `gutter-position` setting.

```
// default gutters
.item { @include gutters; }
```

You can also set explicit gutter widths:

```
// explicit gutters
.item { @include gutters(3em); }
```

Or use the shorthand syntax to adjust settings on the fly:

```
// inside gutters
.item { @include gutters(3em inside); }

// gutters after, in an explicit (10 1/3) layout context
.item { @include gutters(10 1/3 after); }
```

Container

function/mixin

Format `container($layout)`

$layout *<layout>*

Use the `container` **function** to return a container-width based on an optional layout argument, or your global settings.

```
// global settings
width: container();

// 12-column grid
$large-breakpoint: container(12);
```

Use the **mixin** to apply container settings to an element directly.

```
body {
  @include container(12 center static);
}
```

Note that *static math* requires a valid *column-width* setting

---

Nested Context

function/mixin

Function `nested($span)`

Mixin `nested($span) { @content }`

$span **

@content Sass content block

Sass is not aware of the DOM (Document Object Model), or the specific markup of your site, so Susy mixins don't know about any ancestor/child relationships. If your container creates a grid context that is different from the default, you will need to pass that new context explicitly to nested elements.

You can pass that context along with the shorthand syntax.

```
body { @include container(8); }
.span { @include span(3 of 8); }
```

But that gets repetitive if you have large blocks of code using a given context. The `nested` **mixin** provides a shortcut to change the default context for a section of code.

```
@include nested(8) {
  .span { @include span(3); }
}
```

Context is a bit more complex when you are using asymmetrical grids, because we need to know not just *how many* columns, but *which* columns are available.

```
.outer {
  @include span(3 of (1 2 3 2 1) at 2);
```

```
  // context is now (2 3 2)...
  .inner { @include span(2 of (2 3 2) at 1); }
}
```

The `nested` **function** can help you manage context more easily, without having to calculate it yourself.

```
$grid: (1 2 3 2 1);

.outer {
  $context: 3 of $grid at 2;
  @include span($context);

  @include nested($context) {
    .inner { @include span(2 at 1); }
  }
}
```

Global Box Sizing

Set the `box-sizing` on a global selector, and set the *global-box-sizing* to match.

mixin

> **Format** `global-box-sizing($box [, $inherit])`
>
> **Shortcut** `border-box-sizing([$inherit])`
>
> **$box** `content-box | border-box`
>
> **$inherit** [optional] `true | false`

Setting the optional argument, `$inherit`, to `true` will still globally set the `box-sizing`, but in a way such that a component can easily override the global `box-sizing` by setting its own `box-sizing` property. By setting `box-sizing` once on the component, all nested elements within the component will also be modified. The default behavior, where `$inherit` is `false`, would only update the `box-sizing` of the component itself. Nested elements are not affected when `$inherit` is `false`.

You can pass a `box-sizing` argument to the `span` mixin as part of the shorthand syntax, and Susy will set the element's box-sizing to match.

```
// input
.item { @include span(25em border-box); }

// sample output (depending on settings)
.item {
  float: left;
  width: 25em;
  box-sizing: border-box;
}
```

We highly recommend using a global `border-box` setting, especially if you are using inside gutters of any kind.

```
// the basics with default behavior:
* { box-sizing: border-box; }

// the basics with $inherit set to true:
html { box-sizing: border-box; }
* { box-sizing: inherit; }
```

Susy needs to know what box model you are using, so the best approach is to set global box sizing using one of Susy's shortcuts.

```
// the flexible version:
@include global-box-sizing(border-box);

// the shortcut:
@include border-box-sizing;
```

If you want to change the global box-sizing by hand, or it has already been changed by another library, update the *global-box-sizing* setting to let Susy know.

If you need to supprot IE6/7, there is a simple polyfill to make it work.

Rows & Edges

Floated layouts sometimes require help maintaining rows and edges.

Break

mixin

> **Format** `break()`
>
> **Reset** `nobreak()`
>
> **Keywords** `break|nobreak`

To create a new row, you need to clear all previous floats. This can usually be done using keywords with the *span mixin*. When you need to apply a row-break on it's own, we have a `break` mixin.

```
.new-line { @include break; }
```

If you ever need to override that, you can use `nobreak` to set `clear: none;`.

```
.no-new-line { @include nobreak; }
```

Both `break` and `nobreak` can also be used as keywords with the *span mixin*.

First

mixin

> **Format** `first($context)`
>
> **Alternate** `alpha($context)`
>
> **$context** *<layout>*

Note: Only useful when *gutter-position* is set to `before`.

When *gutter-position* is set to `before` we need to remove the gutter from the first element in every row. This can often be solved using a keyword in the *span mixin*. Sometimes you need to set an item as `first` outside the span mixin.

```
.first { @include first; }
```

We also support an `alpha` mixin with the same syntax and output.

Both `first` and `alpha` can also be used as keywords with the *span mixin*.

Last

mixin

Format `last($context)`

Alternate `omega($context)`

$context *<layout>*

Note: Only required when *gutter-position* is set to `after`, but can be useful in any context to help with sub-pixel rounding issues.

When *gutter-position* is set to `after` we need to remove the gutter from the last element in every row, and *optionally float in the opposite direction*. This can often be solved using a keyword in the *span mixin*. Sometimes you need to set an item as `last` outside the span mixin.

```
.last { @include last; }
```

We also support an `omega` mixin with the same syntax and output.

Both `last` and `omega` can also be used as keywords with the *span mixin*.

Full

mixin

Format `full($context)`

$context *<layout>*

This is a shortcut for `span(full)`, used to create elements that span their entire context.

```
.last { @include full; }
```

`full` can also be used as a keyword with the *span mixin*.

Margins

Shortcut mixins for applying left/right margins.

Pre

mixin

Format `pre($span)`

Alternate `push($span)`

$span **

Add margins before an element, depending on the *flow* direction.

```
.example1 { @include pre(25%); }
.example2 { @include push(25%); }
.example3 { @include pre(2 of 7); }
.example4 { @include push(2 of 7); }
```

Post

mixin

Format `post($span)`

$span **

Add margins after an element, depending on the *flow* direction.

```
.example1 { @include post(25%); }
.example2 { @include post(2 of 7); }
```

Pull

mixin

Format `pull($span)`

$span **

Add negative margins before an element, pulling it against the direction of *flow*.

```
.example1 { @include pull(25%); }
.example2 { @include pull(2 of 7); }
```

Squish

mixin

Format `squish($pre [, $post])`

$pre **

$post [optional] **

Shortcut for adding both *pre* and *post* margins to the same element.

```
// equal pre and post
.example1 { @include squish(25%); }

// distinct pre and post
.example2 { @include squish(1, 3); }
```

When they share identical context, you can pass `pre` and `post` spans in the same argument. This is often the case, and saves you from repeating yourself.

```
// shared context
.shared {
  @include squish(1 3 of 12 no-gutters);
}

// distinct context
.distinct {
  @include squish(1 at 2, 3 at 6);
}
```

Padding

Shortcut mixins for applying left/right padding.

Note: The interaction between padding and width changes depending on your given *box-model*. In the browser-default *content-box* model, width and padding are added together, so that an item with `span(3)` and `prefix(2)` will occupy a total of 5 columns. In the recommended *border-box* model, padding is subtracted from the width, so that an item with `span(3)` will always occupy 3 columns, no matter what padding is applied.

Prefix

mixin

Format `prefix($span)`

$span **

Add padding before an element, depending on the *flow* direction.

```
.example1 { @include prefix(25%); }
.example2 { @include prefix(2 of 7); }
```

Suffix

mixin

Format `suffix($span)`

$span **

Add padding after an element, depending on the *flow* direction.

```
.example1 { @include suffix(25%); }
.example2 { @include suffix(2 of 7); }
```

Pad

mixin

Format `pad($prefix [, $suffix])`

$prefix **

$suffix **

Shortcut for adding both *prefix* and *suffix* padding to the same element.

```
// equal pre and post
.example1 { @include pad(25%); }

// distinct pre and post
.example2 { @include pad(1, 3); }
```

When they share identical context, you can pass `pre` and `post` spans in the same argument. This is often the case, and saves you from repeating yourself.

```
// shared context
.shared {
  @include pad(1 3 of 12 no-gutters);
}

// distinct context
.distinct {
  @include pad(1 at 2, 3 at 6);
}
```

Bleed

mixin

Format `bleed($bleed)`

$bleed TRBL (Top Right Bottom Left) **

Apply negative margins and equal positive padding, so that element borders and backgrounds "bleed" outside of their containers, without the content be affected.

This uses the standard *span shorthand*, but takes anywhere from one to four widths, using the common TRBL pattern from CSS.

```
// input
.example1 { @include bleed(1em); }
.example2 { @include bleed(1em 2 20px 5% of 8 .25); }

// output
.example1 {
  margin: -1em;
  padding: 1em;
}

.example2 {
  margin-top: -1em;
  padding-top: 1em;
  margin-right: -22.5%;
  padding-right: 22.5%;
  margin-bottom: -20px;
  padding-bottom: 20px;
  margin-left: -5%;
  padding-left: 5%;
}
```

When possible, the `bleed` mixins will attempt to keep gutters intact. Use the `no-gutters` keyword to override that behavior.

Bleed-x

mixin

Format `bleed-x($bleed)`

$bleed LR (Left Right) **

A shortcut for applying only left and right (horizontal) bleed.

```
// input
.example { @include bleed-x(1em 2em); }

// output
.example {
  margin-left: -1em;
  padding-left: 1em;
  margin-right: -2em;
  padding-right: 2em;
}
```

Bleed-y

mixin

Format `bleed-y($bleed)`

$bleed TB (Top Bottom) **

A shortcut for applying only top and bottom (vertical) bleed.

```
// input
.example { @include bleed-y(1em 2em); }

// output
.example {
  margin-top: -1em;
  padding-top: 1em;
  margin-bottom: -2em;
  padding-bottom: 2em;
}
```

Isolate

mixin

> **Format** `isolate($isolate)`
>
> **$isolate** **

Isolation is a layout technique based on floats, but adjusted to address sub-pixel rounding issues. Susy supports it as a global *output* setting, or as a *Shorthand* keyword for the `span` mixin, or as a stand-alone mixin.

The `$isolate` argument takes a standard *span shorthand*, but any length or grid-index given is interpreted as an isolation location (unless location is otherwise specified with the `at` flag). The function returns a length value.

```
// input
.function {
  margin-left: isolate(2 of 7 .5 after);
}

// output
.function {
  margin-left: 15%;
}
```

And the mixin returns all the properties required for isolation.

```
// input
.mixin { @include isolate(25%); }

// output
.mixin {
  float: left;
  margin-left: 25%;
  margin-right: -100%;
}
```

Gallery

mixin

> **Format** `gallery($span, $selector)`
>
> **$span** **

$selector (nth-) `child`* (default) | `of-type`

Gallery is a shortcut for creating gallery-style layouts, where a large number of elements are laid out on a consistent grid. We take the standard *span shorthand* and apply it to all the elements, using `nth-child` or `nth-of-type` selectors and the isolation technique to arrange them on the grid.

```
// each img will span 3 of 12 columns,
// with 4 images in each row:
.gallery img {
  @include gallery(3 of 12);
}
```

Warning: The *nth-child* selector has issues on iOS8 Safari. Use *nth-of-type* for iOS8 support.

Show Grid

mixin

Format `show-grid($grid)`

$grid *<layout>*

The easiest way to show you grids is by adding a *keyword* to your *container* mixin. If you need to apply the grid separately, the `show-grid` mixin takes exactly the same *layout shorthand* arguments, and can output the debugging grid image as either a background, or a triggered overlay.

```
body {
  @include container;
  @include show-grid(overlay);
}
```

Warning: Grid images are not exact. Browsers have extra trouble with sub-pixel rounding on background images. These are meant for rough debugging, not for pixel-perfect measurements. Expect the `to` side of your grid image (`right` if your flow is `ltr`) to be off by several pixels.

Breakpoint

Susy has built-in media-query handling, and also supports integration with the Breakpoint plugin. To install Breakpoint, follow the instuctions on their site.

Susy Breakpoint

mixin

Format `susy-breakpoint($query, $layout, $no-query)`

$query media query shorthand (see *susy-media*)

$layout *<layout>*

$no-query <boolean> | <string> (see *susy-media*)

`susy-breakpoint()` acts as a shortcut for changing layout settings at different media-query breakpoints, using either *susy-media* or the third-party Breakpoint plugin.

If you are using the third-party plugin, see Breakpoint: Basic Media Queries and Breakpoint: No Query Fallbacks for details.

This mixin acts as a wrapper, adding media-queries and changing the layout settings for any susy functions or mixins that are nested inside.

```
@include susy-breakpoint(30em, 8) {
  // nested code uses an 8-column grid,
  // starting at a 30em min-width breakpoint...
  .example { @include span(3); }
}
```

Susy Media

mixin

Format `susy-media($query, $no-query)`

$query <min-width> [<max-width>] | <string> | <pair> | <map>

$no-query <boolean> | <string>

The `susy-media` mixin provides basic media-query handling, and handles the built-in functionality for `susy-breakpoint`.

$query A single length will be used as a *min-width* query, two lengths will become *min-* and *max-* width queries, a property-value pair, or map of pairs will become `(property: value)` queries, and a lonely string will be used directly.

```
// min
// ---
@include susy-media(30em) { /*...*/ }

@media (min-width: 30em) { /*...*/ }

// min/max pair
// ------------
@include susy-media(30em 60em) { /*...*/ }

@media (min-width: 30em) and (max-width: 60em) { /*...*/ }

// property/value pair
// -------------------
@include susy-media(min-height 30em) { /*...*/ }

@media (min-height: 30em) { /*...*/ }

// map
// ---
@include susy-media((
  min-height: 30em,
  orientation: landscape,
)) { /*...*/ }
```

```
@media (min-height: 30em) and (orientation: landscape) { /*...*/ }
```

$no-query `true` will render the contents to css without any media-query. This can be useful for creating separate no-query fallback files.

For inline fallbacks using a target class, pass in a string (e.g. *.no-mqs*) to use as your fallback selector. The contents will be output both inside a media-query and again inside the given selector.

This can be set globally with the `$susy-media-fallback` variable.

`susy-media` also supports named media-queries, which can be set using the `$susy-media` variable:

```
$susy-media: (
  min: 20em,
  max: 80em 60em,
  string: 'screen and (orientation: landscape)',
  pair: min-height 40em,
  map: (
    media: screen,
    max-width: 30em
  ),
);

@include susy-media(min);
```

Susy One

This is documentation for the old syntax, used in Susy 1.

If you are using Susy 2 and want use the old syntax, change your import from `susy` to `susyone`.

```
// With Susy 2 installed...
@import "susyone";
```

Basic Settings

- **Container**: The root element of a *Grid*.
- **Layout**: The total number of *Columns* in a grid.
- **Grid Padding**: Padding on the sides of the *Grid*.
- **Context**: The number of *Columns* spanned by the parent element.
- **Omega**: Any *Grid Element* spanning the last *Column* in its *Context*.

Total Columns

The number of Columns in your default Grid Layout.

```
// $total-columns: <number>;
$total-columns: 12;
```

- `<number>`: Unitless number.

Column Width

The width of a single Column in your Grid.

```
// $column-width: <length>;
$column-width: 4em;
```

- `<length>`: Length in any unit of measurement (em, px, %, etc).

Gutter Width

The space between Columns.

```
// $gutter-width: <length>;
$gutter-width: 1em;
```

- `<length>`: Units must match `$column-width`.

Grid Padding

Padding on the left and right of a Grid Container.

```
// $grid-padding: <length>;
$grid-padding: $gutter-width;  // 1em
```

- `<length>`: Units should match the container width (`$column-width` unless `$container-width` is set directly).

Functions

Where a mixin returns property/value pairs, functions return simple values that you can put where you want, and use for advanced math.

Columns

Similar to `span-columns` mixin, but returns the math-ready `%` multiplier.

```
// columns(<$columns> [, <$context>, <$style>])
.item { width: columns(3,6); }
```

- `<$columns>`: The number of *Columns* to span,
- `<$context>`: The *Context*. Default: `$total-columns`.
- `<$style>`: Optionally return `static` lengths for grid calculations. Default: `$container-style`.

Gutter

The `%` width of one gutter in any given context.

```
// gutter([<$context>, <$style>])
.item { margin-right: gutter(6) + columns(3,6); }
```

- <$context>: The *Context*. Default: $total-columns.
- <$style>: Optionally return static lengths for grid calculations. Default: $container-style.

Space

Total % space taken by Columns, including internal AND external gutters.

```
// space(<$columns> [, <$context>, <$style>])
.item { margin-right: space(3,6); }
```

- <$columns>: The number of *Columns* to span,
- <$context>: The *Context*. Default: $total-columns.
- <$style>: Optionally return static lengths for grid calculations. Default: $container-style.

Basic Mixins

Container

Establish the outer grid-containing element.

```
// container([$<media-layout>]*)
.page { @include container; }
```

- <$media-layout>: Optional media-layout shortcuts (see *Responsive Grids* below). Default: $total-columns.

Span Columns

Align an element to the Susy Grid.

```
// span-columns(<$columns> [<omega> , <$context>, <$padding>, <$from>, <$style>])
nav { @include span-columns(3,12); }
article { @include span-columns(9 omega,12); }
```

- <$columns>: The number of *Columns* to span. - <omega>: Optional flag to signal the last element in a row.
- <$context>: Current nesting *Context*. Default: $total-columns.
- <$padding>: Optional padding applied inside an individual grid element. Given as a length (same units as the grid) or a list of lengths (from-direction to-direction). Default: false.
- <$from>: The origin direction of your document flow. Default: $from-direction.
- <$style>: Optionally return static lengths for grid calculations. Default: $container-style.

Omega

Apply to any omega element as an override.

```
// omega([<$from>])
.gallery-image {
  @include span-columns(3,9); // each gallery-image is 3 of 9 cols.
  &:nth-child(3n) { @include omega; } // every third image completes a row.
}
```

- <$from>: The origin direction of your document flow. Default: $from-direction.

Nth-Omega

Apply to any element as an nth-child omega shortcut. Defaults to :last-child.

```
// nth-omega([<$n>, <$selector>, <$from>])
.gallery-image {
  @include span-columns(3,9); // each gallery-image is 3 of 9 cols.
  @include nth-omega(3n); // same as omega example above.
}
```

- <$n>: The keyword or equation to select: [first | only | last | <equation>]. An equation could be e.g. 3 or 3n or '3n+1'. Note that quotes are needed to keep complex equations from being simplified by Compass. Default: last.
- <$selector>: The type of element, and direction to count from: [child | last-child | of-type | last-of-type]. Default: child.
- <$from>: The origin direction of your document flow. Default: $from-direction.

Responsive Mixins

- **Breakpoint**: A min- or max- viewport width at which to change *Layouts*.
- **Media-Layout**: Shortcut for declaring *Breakpoints* and *Layouts* in Susy.

```
// $media-layout: <min-width> <layout> <max-width> <ie-fallback>;
// - You must supply either <min-width> or <layout>.
$media-layout: 12;           // Use 12-col layout at matching min-width.
$media-layout: 30em;         // At min 30em, use closest fitting layout.
$media-layout: 30em 12;      // At min 30em, use 12-col layout.
$media-layout: 12 60em;      // Use 12 cols up to max 60em.
$media-layout: 30em 60em;    // Between min 30em & max 60em, use closest layout.
$media-layout: 30em 12 60em;// Use 12 cols between min 30em & max 60em.
$media-layout: 60em 12 30em;// Same. Larger length will always be max-width.
$media-layout : 12 lt-ie9;  // Output is included under ''.lt-ie9'' class,
                            // for use with IE conditional comments
                            // on the <html> tag.
```

- <$min/max-width>: Any length with units, used to set media breakpoints.
- <$layout>: Any (unitless) number of columns to use for the grid at a given breakpoint.
- <$ie-fallback>: Any string to use as a fallback class when mediaqueries are not available. Do not include a leading "." class-signifier, only the class name ("lt-ie9", not ".lt-ie9"). This can be anything you want: "no-mediaqueries", "ie8", "popcorn", etc.

At-Breakpoint

At a given min- or max-width Breakpoint, use a given Layout.

```
// at-breakpoint(<$media-layout> [, <$font-size>]) { <@content> }
@include at-breakpoint(30em 12) {
  .page { @include container; }
}
```

- `<$media-layout>`: The *Breakpoint/Layout* combo to use (see above).
- `<$font-size>`: Browsers interpret em-based media-queries using the browser default font size (`16px` in most cases). If you have a different base font size for your site, we have to adjust for the difference. Tell us your base font size, and we'll do the conversion. Default: `$base-font-size`.
- `<@content>`: Nested `@content` block will use the given *Layout*.

Layout

Set an arbitrary Layout to use with any block of content.

```
// layout(<$layout-cols>) { <@content> }
@include layout(6) {
  .narrow-page { @include container; }
}
```

- `<$layout-cols>`: The number of *Columns* to use in the *Layout*.
- `<@content>`: Nested `@content` block will use the given *Layout*.

Set Container Width

Reset the width of a Container for a new Layout context. Can be used when `container()` has already been applied to an element, for DRYer output than using `container` again.

```
// set-container-width([<$columns>, <$style>])
@include container;
@include at-breakpoint(8) {
  @include set-container-width;
}
```

- `<$columns>`: The number of *Columns* to be contained. Default: Current value of `$total-columns` depending on *Layout*.
- `<$style>`: Optionally return `static` lengths for grid calculations. Default: `$container-style`.

With Grid Settings

Use different grid settings for a block of code - whether the same grid at a different breakpoint, or a different grid altogether.

```
// with-grid-settings([$<columns>, $<width>, <$gutter>, <$padding>]) { <@content> }
@include with-grid-settings(12,4em,1.5em,1em) {
  .new-grid { @include container; }
};
```

- `<$columns>`: Overrides the `$total-columns` setting for all contained elements.

- `<$width>`: Overrides the `$column-width` setting for all contained elements.
- `<$gutter>`: Overrides the `$gutter-width` setting for all contained elements.
- `<$padding>`: Overrides the `$grid-padding` setting for all contained elements.
- `<@content>`: Nested `@content` block will use the given grid settings.

Box Sizing

Border-Box Sizing

Set the default box-model to `border-box`, and adjust the grid math accordingly.

```
// border-box-sizing()
@include border-box-sizing;
```

This will apply border-box model to all elements (using the star selector) and set `$border-box-sizing` to `true`. You can use the variable on it's own to adjust the grid math, in cases where you want to apply the box-model separately.

Isolation

Isolate

Isolate the position of a grid element relative to the container. This should be used in addition to `span-columns` as a way of minimizing sub-pixel rounding errors in specific trouble locations.

```
// isolate(<$location> [, <$context>, <$from>, <$style>])
@include span-columns(4); // 4-columns wide
@include isolate(2); // positioned in the second column
```

- `<$location>`: The container-relative column number to position on.
- `<$context>`: Current nesting *Context*. Default: `$total-columns`.
- `<$from>`: The origin direction of your document flow. Default: `$from-direction`.
- `<$style>`: Optionally return `static` lengths for grid calculations. Default: `$container-style`.

Isolate Grid

Isolate a group of elements in a grid (such as an image gallery) using nth-child or nth-of-type for positioning. Provide the column-width of each element, and Susy will determine the positioning for you.

```
// isolate-grid(<$columns> [, <$context>, <$selector>, <$from>, <$style>])
.gallery-item {
  @include isolate-grid(3);
}
```

- `<$columns>`: The number of *Columns* for each item to span.
- `<$context>`: Current nesting *Context*. Default: `$total-columns`.
- `<$selector>`: either 'child' or 'of-type'. Default: `child`.

- <$from>: The origin direction of your document flow. Default: $from-direction.
- <$style>: Optionally return static lengths for grid calculations. Default: $container-style.

Padding Mixins

Prefix

Add Columns of empty space as padding before an element.

```
// prefix(<$columns> [, <$context>, <$from>, <$style>])
.box { @include prefix(3); }
```

- <$columns>: The number of *Columns* to be added as padding before.
- <$context>: The *Context*. Default: $total-columns.
- <$from>: The origin direction of your document flow. Default: $from-direction.
- <$style>: Optionally return static lengths for grid calculations. Default: $container-style.

Suffix

Add columns of empty space as padding after an element.

```
// suffix(<$columns> [, <$context>, <$from>, <$style>])
.box { @include suffix(2); }
```

- <$columns>: The number of *Columns* to be added as padding after.
- <$context>: The *Context*. Default: $total-columns.
- <$from>: The origin direction of your document flow. Default: $from-direction.
- <$style>: Optionally return static lengths for grid calculations. Default: $container-style.

Pad

Shortcut for adding both Prefix and Suffix padding.

```
// pad([<$prefix>, <$suffix>, <$context>, <$from>, <$style>])
.box { @include pad(3,2); }
```

- <$prefix>: The number of *Columns* to be added as padding before.
- <$suffix>: The number of *Columns* to be added as padding after.
- <$context>: The *Context*. Default: $total-columns.
- <$from>: The origin direction of your document flow. Default: $from-direction.
- <$style>: Optionally return static lengths for grid calculations. Default: $container-style.

Bleed

Add negative margins and matching positive padding to an element, so that its background "bleeds" outside its natural position.

```
// bleed(<$width> [<$sides>, <$style>])
@include bleed(2);
```

- `<$width>`: The number of *Columns* or arbitrary length to bleed. Use `2 of 12` syntax for context in nested situations.
- `<$sides>`: The sides of the element that should bleed. Default: `left right`.
- `<$style>`: Optionally return `static` lengths for grid calculations. Default: `$container-style`.

Margin Mixins

Pre

Add columns of empty space as margin before an element.

```
// pre(<$columns> [, <$context>, <$from>, <$style>])
.box { @include pre(2); }
```

- `<$columns>`: The number of *Columns* to be added as `margin` before.
- `<$context>`: The *Context*. Default: `$total-columns`.
- `<$from>`: The origin direction of your document flow. Default: `$from-direction`.
- `<$style>`: Optionally return `static` lengths for grid calculations. Default: `$container-style`.

Post

Add columns of empty space as margin after an element.

```
// post(<$columns> [, <$context>, <$from>, <$style>])
.box { @include post(3); }
```

- `<$columns>`: The number of *Columns* to be added as `margin` after.
- `<$context>`: The *Context*. Default: `$total-columns`.
- `<$from>`: The origin direction of your document flow. Default: `$from-direction`.
- `<$style>`: Optionally return `static` lengths for grid calculations. Default: `$container-style`.

Squish

Shortcut to add empty space as margin before and after an element.

```
// squish([<$pre>, <$post>, <$context>, <$from>, <$style>])
.box { @include squish(2,3); }
```

- `<$pre>`: The number of *Columns* to be added as `margin` before.
- `<$post>`: The number of *Columns* to be added as `margin` after.

- <$context>: The *Context*. Default: $total-columns.
- <$from>: The origin direction of your document flow. Default: $from-direction.
- <$style>: Optionally return static lengths for grid calculations. Default: $container-style.

Push

Identical to pre.

```
// push(<$columns> [, <$context>, <$from>, <$style>])
.box { @include push(3); }
```

Pull

Add negative margins before an element, to pull it against the flow.

```
// pull(<$columns> [, <$context>, <$from>, <$style>])
.box { @include pull(2); }
```

- <$columns>: The number of *Columns* to be subtracted as margin before.
- <$context>: The *Context*. Default: $total-columns.
- <$from>: The origin direction of your document flow. Default: $from-direction.
- <$style>: Optionally return static lengths for grid calculations. Default: $container-style.

Reset Mixins

Reset Columns

Resets an element to default block behaviour.

```
// reset-columns([<$from>])
article { @include span-columns(6); }     // articles are 6 cols wide
#news article { @include reset-columns; } // but news span the full width
                                          // of their container
```

- <$from>: The origin direction of your document flow. Default: $from-direction.

Remove-Omega

Apply to any previously-omega element to reset it's float direction and margins to match non-omega grid elements. Note that unlike omega, this requires a context when nested.

```
// remove-omega([<$context>, <$from>, <$style>])
.gallery-image {
  &:nth-child(3n) { @include remove-omega; } // 3rd images no longer complete rows.
}
```

- <$context>: Current nesting *Context*. Default: $total-columns.
- <$from>: The origin direction of your document flow. Default: $from-direction.
- <$style>: Optionally return static lengths for grid calculations. Default: $container-style.

Remove Nth-Omega

Apply to any previously nth-omega element to reset it's float direction and margins to match non-omega grid elements. Note that unlike omega, this requires a context when nested.

```
// remove-nth-omega([<$n>, <$selector>, <$context>, <$from>, <$style>])
.gallery-image {
  @include remove-nth-omega(3n); // same as remove-omega example above.
}
```

- `<$n>`: The keyword or equation to select: `[first | only | last | <equation>]`. An equation could be e.g. `3` or `3n` or `'3n+1'`. Note that quotes are needed to keep a complex equation from being simplified by Compass. Default: `last`.
- `<$selector>`: The type of element, and direction to count from: `[child | last-child | of-type | last-of-type ]`. Default: `child`.
- `<$context>`: Current nesting *Context*. Default: `$total-columns`.
- `<$from>`: The origin direction of your document flow. Default: `$from-direction`.
- `<$style>`: Optionally return `static` lengths for grid calculations. Default: `$container-style`.

Debugging

Susy Grid Background

Show the Susy Grid as a background-image on any container.

```
// susy-grid-background();
.page { @include susy-grid-background; }
```

- If you are using the `<body>` element as your *Container*, you need to apply a background to the `<html>` element in order for this grid-background to size properly.
- Some browsers have trouble with sub-pixel rounding on background images. Use this for checking general spacing, not pixel-exact alignment. Susy columns tend to be more accurate than gradient grid-backgrounds.

Container Override Settings

Container Width

Override the total width of your grid with an arbitrary length.

```
// $container-width: <length> | <boolean>;
$container-width: false;
```

- `<length>`: Length in em, px, %, etc.
- `<boolean>`: True or false.

Container Style

Override the type of shell containing your grid.

```
// $container-style: <style>;
$container-style: magic;
```

- `<style>`: `magic | static | fluid`.
  - `magic`: Susy's magic grid has a set width, but becomes fluid rather than overflowing the viewport at small sizes.
  - `static`: Susy's static grid will retain the width defined in your settings at all times.
  - `fluid`: Susy's fluid grid will always be based on the viewport width. The percentage will be determined by your grid settings, or by `$container-width`, if either is set using `%` units. Otherwise it will default to `auto` (100%).

Direction Override Settings

From Direction

The side of the Susy Grid from which the flow starts. For ltr documents, this is the left.

```
// $from-direction: <direction>;
$from-direction: left;
```

- `<direction>`: `left | right`

Omega Float

The direction that Omega elements should be floated.

```
// $omega-float: <direction>;
$omega-float: opposite-position($from-direction);
```

- `<direction>`: `left | right`

Compass Options

Base Font Size

From the Compass Vertical Rhythm module, Susy uses your base font size to help manage em-based media-queries.

```
// $base-font-size: <px-size>;
$base-font-size: 16px;
```

- `<px-size>`: Any length in `px`. This will not actually effect your font size unless you use other Vertical Rhythm tools, we just need to know. See Compass Docs for further usage details.

Browser Support

Susy recognizes all the Compass Browser Support variables, although only IE6 and IE7 have special cases attached to them currently.

```
// $legacy-support-for-ie  : <boolean>;
// $legacy-support-for-ie6 : <boolean>;
// $legacy-support-for-ie7 : <boolean>;
$legacy-support-for-ie  : true;
$legacy-support-for-ie6 : $legacy-support-for-ie;
$legacy-support-for-ie7 : $legacy-support-for-ie;
```

- <boolean>: true | false

Breakpoint Output

If you are compiling seperate files for IE-fallbacks, it can be useful to output only the modern code in one file and only the fallbacks in another file. You can make `at-breakpoint` do exactly that by using the following settings.

$breakpoint-media-output

Turn off media-query output for IE-only stylesheets.

```
// $breakpoint-media-output: <boolean>;
$breakpoint-media-output: true;
```

- <boolean>: true | false

$breakpoint-ie-output

Turn off media-query fallback output for non-IE stylesheets.

```
// $breakpoint-ie-output: <boolean>;
$breakpoint-ie-output: true;
```

- <boolean>: true | false

$breakpoint-raw-output

Pass through raw output without media-queries or fallback classes for IE-only stylesheets.

```
// $breakpoint-raw-output: <boolean>;
$breakpoint-raw-output: false;
```

- <boolean>: true | false

Upgrade Path

Susy `2.x` supports two syntax options, side by side. If you want to use the latest release but keep the old syntax, change your import from `susy` to `susyone`.

```
// With Susy 2.x installed...
@import "susyone";
```

If you ever want to upgrade to the new syntax, change the import back to `susy`, and follow these instructions:

Settings

In SusyOne, settings were handled as variables.

```
// the basics
$columns: 12;
$column-width: 4em;
$gutter-width: 1em;
$grid-padding: 1em;

// advanced
$container-width: false;
$container-style: magic;
$from-direction: left;
$omega-float: right;

// media-query fallbacks
$breakpoint-media-output: true;
$breakpoint-ie-output: true;
$breakpoint-raw-output: false;
```

Removed Settings

All the media-query fallback settings have been dropped. Susy 2 no longer manages media-queries, but we play well with other media-query libraries, and include special *Breakpoint integration*. See their documentation for handling legacy browsers.

We've also dropped `$grid-padding` as a setting. If you want to add padding to your container, you can easily do it by hand.

Translation

The remaining settings can be easily mapped into the new syntax.

```
$susy: (
  // the basics
  columns: $total-columns,
  gutters: $gutter-width / $column-width,
  column-width: $column-width,

  // advanced
  container: $container-width,
  math: if($container-style == magic, fluid, $container-style),
  flow: if($from-direction == right, rtl, ltr),
  last-flow: if($omega-float == $from-direction, from, to),
);
```

There are a few differences to note in the translation.

- You can set either `column-width` or `container` (or neither), but never both. One can be calculated from the other, but if you set both we don't know which one should have priority.
- If you are using `static` math we highly recommend that you use `column-width` rather than `container`.
- The old `magic` style can be achieved through a combination of `fluid` math and a width setting (either `column-width` or `container`).

See *Settings* for more details.

Functions

Columns / Space

The `columns` and `space` functions from SusyOne have now been merged into the new *span function*.

```
// columns/space: <span> [, <context>, <math>]
$columns: columns(3, 6, static);
$space: space(2, 7, fluid);

// span
$span-columns: span(3 of 6 static);
$span-space: span(2 of 7 fluid wide);
```

The difference between `columns` and `space` in the old syntax is now covered by the `narrow` and `wide` *spread* keywords (with `narrow` being the default in most cases).

Gutter

The *gutter function* remains, but the syntax has changed.

```
// gutter([<context>, <math>])
$old: gutter(6, static);
$new: gutter(6 static);
```

Basic Mixins

Container

The *container mixin* remains, but media-query support has been removed. For now, at least, you'll have to establish one container at a time inside media-query declarations.

In most (fluid/magic) cases, we set up the container for our largest layout, and let it flex fluidly from there. If you need to change explicit sizes at explicit breakpoints, we recommend using the `container` *function* to override only the width at different breakpoints.

```
// old
body {
  @include container(4, 50em 8, 80em 12);
}

// new (simple)
```

```
body { @include container(12); }

// new (with breakpoint plugin)
body {
  @include container(4);
  @include breakpoint(50em) { max-width: container(8); }
  @include breakpoint(80em) { max-width: container(12); }
}
```

Span Columns

The `span-columns` mixin has been renamed *span*, and has much more power and flexibility. The old `$padding` argument has been removed, but everything else tranlates cleanly. Note that `$from` took `right` or `left` as options, where the new `flow` setting takes `rtl` or `ltr`.

```
// span-columns(<$columns> [<omega> , <$context>, <$padding>, <$from>, <$style>])
.old { @include span-columns(3 omega, 12, $padding, left, static); }
.new { @include span(last 3 of 12 ltr static); }
```

Omega

The *omega mixin* still esists, and should work without any changes. For readability, `omega` can be replaced with `last`, but that's up to you.

`nth-omega` has been deprecated, in favor of `omega` with nth-child selectors.

```
.old { @include nth-omega(last); }
.new:last-child { @include omega; }
```

Responsive Design

At-Breakpoint

Media-query support has been removed from the Susy core, because there are so many more powerful and flexible query-handling plugins. We recommend using Breakpoint, and we've written a translation of `at-breakpoint` (now called *susy-breakpoint*) that integrates smoothly with their controls.

```
.old {
  @include at-breakpoint(30em 8 60em) {
    // your 8-column media-query content...
  }
}

.new {
  @include susy-breakpoint(30em 60em, 8) {
    // your 8-column media-query content...
  }
}
```

This looks like a minor change, but it exposes a lot more power in both the media-queries and the changes to layout. See the Breakpoint docs for more detail on the former, and use our *shorthand* to control the latter in detail.

Layout & With-Grid-Settings

the `layout` and `with-grid-settings` mixins have merged to become *with-layout*. They continue to work much like before, with extra power exposed through the *shorthand* syntax.

```
// old
@include layout(12) { /* your 12-column layout */ }
@include with-grid-settings(8, 4em, 1em) { /* your custom layout */ }

// new
@include with-layout(12) { /* your 12-column layout... */ }
@include with-layout(8 (4em 1em)) { /* your custom layout */ }
```

There is still a mixin named `layout`, but it changes the global layout settings rather than wrapping a layout block.

```
// global layout
@include layout(12);

/* your 12-column layout... */
```

Set Container Width

The `set-container-width` mixin can be replaced by applying the *container function* to the `width` or `max-width` of your containing element.

```
// old
.fluid { @include set-container-width(8, fluid); }
.static { @include set-container-width(12, static); }

// new
.fluid { max-width: container(8); }
.static { width: container(12); }
```

Grid Helpers

Border-Box Sizing

The setting has changed from the boolean `$border-box-sizing` to the new *global-box-sizing*, but the *border-box-sizing* mixin works exactly like before.

Isolate

Isolation no longer requires it's own mixin, as it can be controlled now through the *span mixin* for most cases. In those cases where you do still need a distinct mixin, *isolate* remains much like before.

```
.old { @include isolate(2, 12, left, static); }
.new { @include isolate(2 of 12 ltr static); }
```

Isolate Grid

the `isolate-grid` mixin has been renamed *gallery*, but is very similar in use.

```
.gallery-old { @include isolate-grid(3, 12, child, left, fluid); }
.gallery-new { @include gallery(3 of 12 left fluid, child); }
```

Only the `selector` argument remains split off from the others.

Margins and Padding

All the margin and padding mixins — `pre`, `post`, `push`, `pull`, `prefix`, `suffix`, `pad`, `squish` — remain unchanged, except that we now use the *shorthand syntax* in place of all the arguments.

See the new *margins / padding* documentation for details.

Bleed

Besides upgrading to the new shorthand, the *bleed* mixin now also supports TRBL syntax for applying to different sides, along with `bleed-x` and `bleed-y` shortcuts for horizontal and vertical bleed.

```
.old { @include bleed(2, left right); }
.new { @include bleed-x(2); }
```

Susy Grid Background

This has been renamed *show-grid*, and otherwise remains intact.

Reset-Columns / Remove-Omega

Susy One included `reset-columns` and `remove-omega`, but both have been deprecated. Rather than removing styles, override them with the desired behavior. The *full* and *span* mixins should give you everything you need for overriding spans and omegas, respectively.

DIY Susy

Susy is built in three distinct modules: math, output, and syntax. The math and output layers form the core of Susy — so abstract that they could be used for any grid system. That's exactly what we hope will happen.

The syntax modules hold it all together. In the same way that you can theme a website, applying different CSS to the same markup, you can theme Susy by writing your own syntax (or extending one of ours).

We've written a powerful new *Default Syntax*, and we're keeping the old *Susy One* available as well. But why stop there? You can *create your own unique syntax*, or port over the language of existing tools like oocss, singularity, zurb, neat, zen, blueprint, 960gs, etc., without ever leaving Susy.

Core Settings

While the Susy language module is built to support layouts of all kinds, we only need the math module for grid basics.

The Susy core has two settings: *columns*, and *gutters*.

```
$symmetrical: (
  columns: 12,
  gutters: 1/4,
);

$asymmetrical: (
  columns: (1 3 4 6 2),
  gutters: .5,
);
```

Both `columns` and `gutters` are set as unitless numbers, but you can think of them as "grid units" — as they are all relative to each other. `1/4` gutter is a quarter the size of `1` column.

Is Symmetrical

Returns `null` if a grid is asymmetrical.

- `$columns: <number>|<list>`

It's not a difficult test, but it's important to know what you're dealing with.

```
// input
$sym: is-symmetrical(12);
$asym: is-symmetrical(2 4 6 3);

// output
$sym: 12;
$asym: null;
```

Susy Count

Find the number of columns in a given layout.

- `$columns: <number>|<list>`

This is only necessary for asymmetrical grids, since symmetrical are already defined by their count, but the function handles both styles for the sake of flexibility.

1. `<number>`: Susy grid layouts are defined by columns. In a symmetrical grid all the columns are the same relative width, so they can be defined by the number of columns. We can have an "8-column" grid, or a "12-column" grid.

   ```
   // input
   $count: susy-count(12);

   // output
   $count: 12;
   ```

2. `<list>`: Asymmetrical grids are more complex. Since each column can have a different width relative to the other columns, we need to provide more detail about the columns. We can do that with a list of relative (unitless sizes). Each number in the list represents a number of grid units relative to the other numbers.

```
// input
$count: susy-count(1 2 4 3 1);

// output
$count: 5;
```

For asymmetrical grids, the number of columns is egual to the list length. This isn't complex math.

Column Sum

Find the total sum of column-units in a layout.

- `$columns: <number>|<list>`
- `$gutters: <ratio>`
- `$spread: false/narrow|wide|wider`

Rather than counting how many columns there are, the `susy-sum` function calculates the total number of grid units covered. It's a simple matter of adding together all the columns as well as the gutters between them.

```
// input
$susy-sum: susy-sum(7, .5);

// output: 7 + (6 * .5) = 10
$susy-sum: 10;
```

Most grids have one less gutter than column, but that's not always true. The `spread` argument allows you to also include the gutters on either side. While the default `narrow` spread subtracts a gutter, the `wide` spread (common when using split gutters) has an equal number of columns and gutters.

```
// input
$wide-sum: susy-sum(7, .5, wide);

// output: 7 + (7 * .5) = 10.5
$wide-sum: 10.5;
```

On rare occasions you may actually want gutters on both sides, which we call a `wider` spread.

```
// input
$wider-sum: susy-sum(7, .5, wider);

// output: 7 + (8 * .5) = 11
$wider-sum: 11;
```

This is all possible with asymmetrical grids as well.

```
// input
$susy-sum: susy-sum(1 2 4 2, 1/3);

// output: (1 + 2 + 4 + 2) + (3 * 1/3) = 10
$susy-sum: 10;
```

Column Span

Return a subset of columns at a given location.

- `$span`: `<number>`
- `$location`: `<number>`
- `$columns`: `<number>|<list>`

This is only necessary for asymmetrical grids, since a symmetrical subset is always equal to the span, but the function handles both styles for the sake of flexibility.

The `location` is given as a column index, starting with 1, so that `1` is the first column, `2` is the second, and so on.

```
// input
$sym-span: susy-span(3, 2, 7);
$asym-span: susy-span(3, 2, (1 2 3 5 4));

// output: 3 columns, starting with the second
$sym-span: 3;
$asym-span: (2 3 5);
```

Susy

Find the sum of given slice.

- `$span`: `<number>`
- `$location`: `<number>`
- `$columns`: `<number>|<list>`
- `$gutters`: `<ratio>`
- `$spread`: `false/narrow|wide|wider`

This is where it all comes together. `susy` is the basic building-block for any grid system. It combines `susy-span` with `susy-sum` to return the unitless width of a given slice.

```
// input
$sym-span: susy(3, 2, 7, .5);
$asym-span: susy(3, 2, (1 2 3 5 4), .5);

// output
$sym-span: 4;
$asym-span: 11;
```

All you need to do is add units...

Build Something New

That's really all it takes to build a grid system. The rest is just syntax. Start with `susy()`.

```
$sum: susy(3, 2, 7);
```

If you want static grids, you can multiply the results by the width of one column.

```
// static
$column-width: 4em;
$static: $sum * $column-width;
```

For a fluid grid, divide the results by the context span sum, to get a percentage.

```
// fluid
$context: susy(7);
$fluid: percentage($sum / $context);
```

That's all it takes. Now go build yourself a grid system!

Changelog

2.2.13 - Apr 10 2018

- Support Sass 3.5+ *get-function* requirements.
- Doc updates & typo fixes.

2.2.12 - Jan 25 2016

- Fix bug in validation-errors.

2.2.11 - Jan 15 2016

- Fix bug with *susy-inspect*.

2.2.10 - Jan 7 2016

- Add *$pixel-values-only* setting to SusyOne, for turning off Compass *rem* support.

2.2.6 - Sep 1 2015

- Fix a bug with overlay grids.
- Fix a bug with 0-width split gutters.
- Other small bug fixes.

2.2.5 - May 14 2015

- Pass grid arguments to overlay positioning mixin.

2.2.3 - Apr 28 2015

- Work around libsass fraction bug.

2.2.2 - Jan 23 2015

- Fix bug in npm package.

2.2.1 - Jan 14 2015

- Release npm `susy` package.
- Add global `$susy-media` map for creating named breakpoints.
- Add internal media-query support for `susy-breakpoint` without requiring the Breakpoint plugin.
- `susy-breakpoint` mixin no longer requires `$layout` argument. By default, no changes will be made to your existing layout.
- Update `global-box-sizing` and the legacy `border-box-sizing` mixins to optionally take another argument, `$inherit`. This new argument is a boolean value that defaults to `false`, meaning the behavior of these mixins will not change by default. The default behavior sets all elements to use the specified `box-sizing`, which can only be changed explicitly on a per-element basis. By passing in `$inherit` as `true`, the `box-sizing` is set on the `html` element, and all other elements inherit this property. This means that the `box-sizing` can be changed at the component level and all nested elements will inherit this change. This cascading effect can be prevented by explicitly setting `box-sizing` on the exceptions within the nested context.
- Add `su` import at root level.
- Both `su` and `susy` work with the latest LibSass master branch (3.0.2+). There are a few exceptions:
  - The `susysone` syntax
  - `overlay` grid output
  - The `inherit` option for `global-box-sizing` & `border-box-sizing`

2.1.3 - Jul 16 2014

- Baseline grid image uses *px* instead of *%*.
- Updated Sass dependency to work with 3.4.

2.1.2 - Apr 28 2014

- `first` and `last` keywords output `0` margins instead of `null` so they can be used to override previous span settings.
- Output `:before` / `:after` rather than `::before` / `::after` to support IE8.
- Load Susy paths in Compass if required, otherwise add it to SASS_PATH. [Adrien Antoine]
- Compass 1.0 config no longer needs to `require 'susy'`. Susy is registered with Compass automatically.
- Add `$clean` argument to `layout` and `with-layout` mixins, for creating new layout contexts from a clean slate.

2.1.1 - Mar 13 2014

- Rename core math functions, and prepare for decomposition.
  - `column-count()` => `susy-count()`
  - `column-sum()` => `susy-sum()`
  - `column-span()` => `susy-slice()`
  - `column-span-sum()` => `susy()`
- Add tests for core math validation.

2.0.0 — Mar 10 2014

- New susyone tests for split-columns, is-default-layout, medialayout, columns, relative-width, columns width and nth-of-type (using True).
- Sass 3.3.0 (Maptastic Maple)
- Rename local 2.0 variables that conflict with global susyone settings.
- Susyone container mixin applies full container settings at every breakpoint.

2.0.0.rc.2 — Mar 4 2014

- Fix *templates_path* and compass project templates
- Fix Compass "rem" integration to respect `$rhythm-units` setting.

2.0.0.rc.1 — Feb 7 2014

- Add browser support module with settings to `use-custom` mixins for `background-image`, `background-options` (`-size`, `-clip`, `-origin`), `box-sizing`, `clearfix`, and `rem`. If you set to `false`, we'll make sure everything works well on modern browsers. If you set to `true`, we'll check for existing mixins (e.g. from Compass or Bourbon) to provide more powerful legacy support.
- Fix bugs caused by Sass changes to `str-index()`, `#{&}`, and `@at-root`.
- Fix Bower dependencies, and add support for Sache.
- Remove legacy Compass polyfils from susyone code.

2.0.0.beta.3 — Jan 10 2014

- Fix a bug making `show-grid` unaware of local `debug/output` keywords.
- Added Susyone syntax for those that need to use the old Susy syntax, with updated Sass and Compass.
  - `@import 'susyone';`

2.0.0.beta.2 — Jan 6 2014

- Allow nesting of Susy settings.
- `show-grid` mixin can output either background or overlay grids.
- Add `isolate` function to return isolation offset width.
- Fix a bug with `last` output for `split`-gutter layouts.
- Fix a bug with split-gutter `span()`, and `narrow`/`wider` keywords.
- Fix a bug with `bleed` and `null` + `inside` gutters.
- `bleed` output uses TRBL shorthand when possible.
- Clean up and document the core math functions.
- Document upgrade path, core-math, and DIY grids.

BREAKING:

- Move debug settings into `$susy: (debug: (<settings>));`.
- Replace `show-grid` setting with new `debug: image` setting.
- Add `debug: output` setting and keywords to toggle between `background` and `overlay` grid images.
- Remove `grid-overlay` mixin.
  - Becomes part of `show-grid` mixin.
  - Doesn't take `$selector` argument — should be nested instead.
  - Can still be used multiple times.
- `isolate` mixin now interprets span argument as location, unless location is otherwise specified.
  - `isolate(2)` is the same as `isolate(at 2)`.
  - `isolate(25%)` will isolate *at* `25%`.
- Rename setting controls for consistency.
  - `set-grid` => `layout`
  - `use-grid` => `with-layout`
- `pad` and `squish` use RL shorthand for shared context.
  - `pad(1, 3 of 12)` => `pad(1 3 of 12)`

2.0.0.beta.1 — Dec 24 2013

- Add `susy-breakpoint` mixin for basic integration with Breakpoint.
  - Syntax: `breakpoint($query, $layout, $no-query)` where `$query` and `no-query` follow the Breakpoint syntax, and `$layout` uses the Susy syntax for defining grids.
- Add `layout` function to convert layouts from shorthand syntax to map.
- Add `full` keyword shortcut for full-width spans.
- BREAKING: Remove unclear `row` and `unrow` mixins.
- Add `break` and `nobreak` mixins/keywords to create a new line before any element in the layout.
- BREAKING: Rename `is-container: container` setting/value to `role: nest`.

- BREAKING: Rename `layout-method` setting to `output`.
- BREAKING: Rename `layout-math` setting to `math`.
- Clean up division between math/output/syntax layers.
- `gutters` and `container-position` can be set to `null`.
- If `gutters` are set to `0` or `null`, they will have no output.
- BREAKING: `full` output matches span patterns.
- BREAKING: Debug grids are hidden by default.
- BREAKING: Remove `nth-last/-omega/-first/-alpha` as confusing & out-of-scope. Format your nth-selectors manually to apply `first/last` mixins.
- Gutter mixins/functions can accept context-only (without the "of" syntax):
  - `gutters(of 10 .25)` == `gutters(10 .25)`
  - Unitless numbers are used for context.
  - Lengths (with units) are used as explicit gutter-overrides.
- BREAKING: Re-purposed `susy-set` as reverse of `susy-get` — to adjust a single setting. Example: `@include susy-set(gutter-position, inside);`
- Replace global `box-sizing` setting with `global-box-sizing`.
  - Let Susy know what box model you are using globally.
  - `box-sizing` can still be passed as a keyword argument.
- Add `global-box-sizing()` mixin to set your global box model.
  - Example: `@include global-box-sizing(border-box);`
  - You can still use the legacy `@include border-box-sizing;` as a shortcut.
  - Uses your global setting as a default.
  - Updates your global setting to match, if you pass a different value.
- `gallery` and `span` mixins take global-box-sizing into account.

2.0.0.alpha.6 — Dec 5 2013

- Rewrite syntax parsing so parser and resulting maps are shared across Susy.
- Fix explicit-span bug causing large gutters.
- Padding mixins now respect inside gutters.

Backwards Incompatible:

- Removed `gutters $n` keyword in shorthand syntax for setting explicit gutters. Use `(gutter-override: $n)` map instead.

2.0.0.alpha.5 — Nov 25 2013

- Compass is no longer a dependency.
  - Only registers as a compass extension if compass is present.
- Any mixin/function that accepts natural language syntax also accepts maps.

- Maps and natural language can be mixed:
  - `$large: (columns: 12, gutters: .5);`
  - `span(3 $large no-gutters)`
- Add `full` mixin for full-width spans.

Backwards Incompatible:

- Requires Sass 3.3
- Default settings are handled with a Sass map on the `$susy` variable. Example: `$susy: (columns: 12, gutters: .25)` etc.
- `bleed` now takes standard span syntax, with multiple (TRBL) spans.
  - e.g. `bleed(1em 2 of 8)` for 1em top/bottom and 2-columns left/right.
  - Add `bleed-x`/`bleed-y` mixins for horizontal and vertical shortcuts.
- Span arguments now accept `narrow`, `wide`, or `wider` keywords.
  - The `wide` keyword replaces the old `outer` keyword.
  - This setting has been re-named from `outer` to `spread`.
- Re-wrote grid debugging
  - More concise & accurate output for symmetrical grids.
  - Changed `grid-background()` to `show-grid()`/`show-grids()`
  - Changed `overlay-grid()` to `grid-overlay()`
  - Moved settings into `$debug: (color: rgba(#66f, .25), toggle: top right);`
  - Removed overlay-position setting.
  - Only display vertical-rhythms when `$base-line-height` is available.
- `split` gutters are no longer removed at the grid edges.
  - `first` and `last` are not special cases for split gutter-handling.
  - pass the `container` argument to wrappers you plan to nest inside.
- `first`/`alpha`/`last`/`omega`/`nth-` mixins require grid context.

2.0.0.alpha.4 — Sept 4 2013

- Add `bleed` mixin.
- Fix bug with fluid inside-gutter calculations.
- `$last-flow` setting controls the flow direction of row-ending elements.
- `background-grid-output` now accepts `$line-height` argument.
- Compass modules are imported as needed.
- `grid-background`, `grid-overlay`, `grid-background-output`, & `$grid-background-color` have been renamed to remiain consistent and avoid conflicts with Compass:
  - `grid-background` => `background-grid`
  - `grid-overlay` => `overlay-grid`

  - `grid-background-output` => `background-grid-output`
  - `$grid-background-color` => `$grid-color`
- `span` mixing accepts nested `@content`, and uses nested context.
- Add `inside-static` option for static gutters in otherwise fluid grids.
- `gutters` mixin uses span syntax, accepts explicit gutter span.
- Explicit gutter-overrides are divided when gutters are `split`/`inside`.

2.0.0.alpha.3 — July 9 2013

- `row` now includes clearfix, and `unrow` removes clearfix.
- `gallery` output should override previous gallery settings.
- Removed `nth-gallery` and `isolate-gallery` in favor of single, isolated `gallery` mixin.
- Add padding-span syntax: `prefix`, `suffix`, and `pad`.
- Add margin-span syntax: `pre`, `post`, `push`, `pull`, and `squish`.
- New `gutters` mixin adds gutters to an element.
- `gutter` function now returns half-widths when using split/inside gutters.
- Add `outer` keyword to `span` syntax, to return span-width including gutters.
  - Works with both span mixin and span function.
  - Replaces Susy 1.0 `space` function.
- Add comrehensive unit tests, using True.
- Improve fall-abck handling of ommitted arguments.
- Add `container` function to return a given container's width.
- Add `auto` keyword to override `$container-width`, otherwise respect existing width.
- Renamed `$isolate` to `$layout-method`
  - No longer accepts boolean.
  - Accepts keywords `isolate` and (default) `float`.
- Renamed `$static` to `$layout-math`
  - No longer accepts boolean.
  - Accepts keywords `static` (use given units) and (default) `fluid` (use % units).
- Add `show-columns` and `show-baseline` keywords to `$show-grids` setting. `show` will show both columns/baseline, default is `show-columns`.

2.0.0.alpha.2 — May 7 2013

- Added `gutter <length>`/`gutters <length>` to override the attached gutter width on a single span. NOTE: `gutters 0` is not the same as `no-gutters`. `0` is an output value, `no-gutters` removes output.
- Added `container` span option to remove inside gutters from nesting containers.
- Added `before`/`after`/`split`/`inside`/`no-gutters` gutter options.

- Added `gallery` mixin for auto-generating gallery layouts.
- Moved grid-backgrounds into language layer, and made them syntax-aware.
- Added `row`/`unrow`, `first`/`last`, `alpha`/`omega`, `nth-first`/`nth-last`, and `nth-alpha`/`nth-omega`.
- Added `container` and `span` mixins with new syntax.
- Added syntax-aware math functions (`span`/`gutter`/`outer-span`).
- Added rough `translate-susy1-settings` mixin.
- Moved syntax-specific math into language layer.
- Fleshed-out new language syntax.
- Added `get-grid`, `set-grid`, and `use-grid` and declaring and managing settings.
- Remove breakpoint core requirement (will come back as option)

2.0.0.alpha.1 — Jan 26 2013

Susy 2.0 was re-written from the ground up.

- Functioning math engine
- Initial string parsing for natural syntax
- Float/Isolation output methods
- Removed all ECHOE/RAKE stuff in favor of vanilla .gemspec
- Added Ruby based String Split function
- Added Sass based `grid-add` function, to add grids à la Singularity
- Added default variables

1.0.5 — Nov 27 2012

- Add support for rem-units.
- Clean-up quoted arguments.
- Fix a few bugs related to the override settings.

1.0.4 — Nov 3 2012

- Fix bug in nested mixins that adjust support (e.g. `nth-omeg` inside `at-breakpoint`).
- Remove non-ie experimental support in `at-breakpoint` ie-fallback output.

1.0.3 — Oct 20 2012

- Fix Compass dependencies.

1.0.2 — Oct 20 2012

- Fix a bug with `container-outer-width` ignoring `$columns` argument.
- Turn off legacy-ie support inside CSS3 selectors (`nth-omega` etc).

1.0.1 — Sept 12 2012

- Fix a bug in the relationship between `$container-width` and `$border-box-sizing`, so that grid-padding is subtracted from the width in certain cases.
- Reset right margin to `auto` rather than `0` with `remove-omega`.

1.0 — Aug 14 2012

This release is loaded with new features, but don't let that fool you. Susy just became shockingly simple to use.

The gem name has changed from `compass-susy-plugin` to `susy`. First uninstall the old gem, then install the new one. If you have both gems installed, you will get errors.

Semantics:

We re-arranged the code in order to make the syntax simpler and more consistent:

- `$total-cols` is now `$total-columns`.
- `$col-width` is now `$column-width`.
- `$side-gutter-width` is now `$grid-padding` and gets applied directly to the grid container.
- `un-column` & `reset-column` mixins have merged into `reset-columns`.
- `columns` has been renamed `span-columns` to resolve the conflict with CSS3 columns. See other improvements to span-columns below.

We also removed several bothersome requirements:

- The `alpha` mixin is no longer needed. Ever.
- The `omega` no longer takes a `$context` argument.
- `full` has been removed entirely. Elements will be full-width by default. You can add `clear: both;` back in as needed.
- `side-gutter()` is no longer needed. You can use the `$grid-padding` setting directly.

Upgrade:

That's all you need in order to upgrade from Susy 0.9.

1. Uninstall and re-install the gem.
2. Find and replace the semantic changes listed above.

You're done! Stop worrying about all that "nested vs. root" bullshit, and start playing with the new toys!

If you use the `$from` directional arguments directly in the `span-columns` mixin, there may be one more change to make. See below:

New Features:

- `span-columns` supports new features:
  - "omega" can be applied directly through the `$columns` argument.

  - Internal padding can be added through the `$padding` argument.
  - This pushes the `$from` argument from third position into fourth.

- `at-breakpoint` allows you to change layouts at media breakpoints.
- `container` accepts multiple media-layout combinations as a shortcut.
- `layout` allows you to use a different layout at any time.
- `with-grid-settings` allows you to change any or all grid settings.
- `set-container-width` does what it says, without the other container code.
- `$breakpoint-media-output`, `$breakpoint-ie-output`, and `$breakpoint-raw-output` settings help manage the different outputs from `at-breakpoint` when you have IE-overrides living in a file of their own.
- `border-box-sizing` will apply the popular `* { box-sizing: border-box }` universal box-model fix, as well as changing the Susy `$border-box-model` setting for you, so Susy knows to adjust some math.
- The `space()` function can be used anywhere you need column+gutter math.
- `push`/`pull`/`pre`/`post`/`squish` mixins help manage margins.
- use the `nth-omega` mixin to set omega on any nth-child, nth-of-type, first, last, or only element.
- `remove-omega` and `remove-nth-omega` will remove the omega-specific styles from an element.
- `$container-width` will override the width of your container with any arbitrary length.
- `$container-style` will override the type of grid container (magic, fluid, fixed, static, etc) to use.

0.9 — Apr 25 2011

Everything here is about simplicity. Susy has scaled back to it's most basic function: providing flexible grids. That is all.

Deprecated:

- The `susy/susy` import is deprecated in favor of simply importing `susy`.
- The `show-grid` import is deprecated in favor of CSS3 gradient-based grid-images. You can now use the `susy-grid-background` mixin. See below.

Removed:

- Susy no longer imports all of compass.
- Susy no longer establishes your baseline and no longer provides a reset. All of that is in the Compass core. You can (and should!) keep using them, but you will need to import them from compass.

New:

- Use `susy-grid-background` mixin on any `container` to display the grid. This toggles on and off with the same controls that are used by the compass grid-background module.

0.9.beta.3 — Mar 16 2011

Deprecated:

- The `susy/reset` import has been deprecated in favor of the Compass core `compass/reset` import.

- The `susy` mixin has been deprecated. If you plan to continue using vertical-rhythms, you should replace it with the `establish-baseline` mixin from the Compass Core.

Removed:

- The `vertical-rhythm` module has moved into compass core. The API remains the same, but if you were importing it directly, you will have to update that import. (`$px2em` was removed as part of this, but didn't make it into core).
- The `defaults` template has been removed as 'out-of-scope'. This will not effect upgrading in any way, but new projects will not get a template with default styles.

New Features:

- Susy now supports RTL grids and bi-directional sites using the `$from-direction` variable (default: left) and an optional additional from-direction argument on all affected mixins. Thanks to @bangpound for the initial implementation.
- Susy is now written in pure Sass! No extra Ruby functions included! Thanks to the Sass team for making it possible.

0.8.1 — Sep 24 2010

- Fixed typos in tutorial and `_defaults.scss`

0.8.0 — Aug 13 2010

Deprecated:

- The `skip-link` was deprecated as it doesn't belong in Susy.
- All the IE-specific mixins have been deprecated, along with the `$hacks` variable. Hacks are now used in the default mixins as per Compass.
- The `hide` mixin was deprecated in favor of the Compass `hide-text` mixin.

Other Changes:

- `inline-block-list` will be moved to the compass core soon. In preparation, I've cleaned it up some. You can now apply a padding of "0" to override previous padding arguments. You can also use `inline-block-list-container` and `inline-block-list-item` as you would with the Compass `horizontal-list` mixins.
- The `$align` arguments have been removed from both the `susy` and `container` mixins. Text-alignment is no longer used or needed in achieving page centering. That was a cary-over from the IE5 Mac days.
- The `container` mixin now uses the `pie-clearfix` compass mixin to avoid setting the overflow to hidden.
- Default styles have been cleaned up to account for better font stacks and typography, html5 elements, vertically-rhythmed forms, expanded print styles, use of `@extend`, and overall simplification.

0.7.0 — Jun 01 2010

- updated documentation

0.7.0.rc2 — May 13 2010

- Fixes a bug with grid.png and a typo in the readme. Nothing major here.

0.7.0.rc1 — May 12 2010

- template cleanup & simplification - no more pushing CSSEdit comments, etc.
- expanded base and defaults with better fonts & styles out-of-the-box
- expanded readme documentation. This will expand out into a larger docs/tutorial site in the next week.

0.7.0.pre8 — Apr 20 2010

- mostly syntax and gem cleanup
- added `un-column` mixin to reset elements previously declared as columns.
- added `rhythm` mixin as shortcut for leaders/trailers. accepts 4 args: leader, padding-leader, padding-trailer, trailer.
- added a warning on `alpha` to remind you that `alpha` is not needed at nested levels.

0.7.0.pre7 — Apr 13 2010

- *Requires HAML 3 and Compass 0.10.0.rc2*
- Internal syntax switched to scss. This will have little or no effect on users. You can still use Susy with either (Sass/Scss) syntax.
- `$default-rhythm-border-style` overrides default rhythm border styles
- Better handling of sub-pixel rounding for IE6

0.7.0.pre6 — Mar 29 2010

- Added `+h-borders()` shortcut for vertical_rhythm `+horizontal-borders()`
- Fixed vertical rhythm font-size typo (thanks @oscarduignan)
- Added to template styles, so susy is already in place from the start

0.7.0.pre5 — Mar 19 2010

- Expanded and adjusted `_vertical_rhythm.sass` in ways that are not entirely backwards compatible. Check the file for details.
- `_defaults.sass` is re-ordered from inline to block.
- `:focus` defaults cleaned up.
- README and docs updated.

0.7.0.pre4 — Jan 20 2010

Update: pre2 was missing a file in the manifest. Use pre4.

Update: Forgot to note one change: `+susy` is no longer assigned to the `body` tag, but instead at the top level of the document (not nested under anything).

Warning: This update is not backwards compatible. We've changed some things. You'll have to change some things. Our changes were fairly major in cleaning up the code - yours will be minor and also clean up some code.

Added:

- new `_vertical_rhythm.sass` (thanks to Chris Eppstein) provides better establishing of the baseline grid, as well as mixins to help you manage it.
- `!px2em` has replaced `px2em()` - see below.

Removed:

- `px2em()` has been removed and replaced with a simple variable `!px2em` which returns the size of one pixel relative to your basic em-height. Multiply against your desired px dimensions (i.e. `border-width = !px2em*5px` will output the em-equivalent of 5px).
- `!base_font_size_px` and `!base_line_height_px` have been replaced with `!base_font_size` and `!base_line_height` which take advantage of sass's built-in unit handling.
- `!grid_units` is not needed, as you can now declare your units directly in the other grid `_width` variables. Use any one type of units in declaring your grid. The units you use will be used in setting the container size.

Once you've upgraded, before you compile your files, make these changes:

- remove the "_px" from the font-size and line-height variables, and add "px" to their values.
- remove the `!grid_units` variable and add units to your grid variable values.
- find any uses of `px2em()` and replace them with something.
- enjoy!

0.7.0.pre1 — Nov 30 2009

Not a lot of new functionality here – it all moved over to Compass 0.10.0 – mostly just cleaning it up to match.

- simplified the default styles and gave them their own project template (`_defaults.sass`).
- defaults not imported by `ie.sass`, as `ie.sass` should be cascading on top of `screen.sass` anyway
- changed the syntax to match CSS and Compass (`property:` replaces `:property`)
- added more inline documentation and brought tutorial up to date
- moved CSS3 module over to Compass
- import the compass HTML5 reset along with the normal reset by default (because Susy loves the future)
- little internal management fixes and so on and so on...

Older

Not documented here. Check the commit log...

CHAPTER 2

ToDo

These are the features we're working on next:

- Add IE support to new syntax.
- Move SusyOne syntax onto new math/output modules.
- Add padding/margin options to the `span` mixin, for simpler output.

We're always happy to hear your ideas as well. Leave us a note on GitHub Issues, or fork our code, and submit a pull request!

Note: This isn't neverland, and Susy isn't magic. We're still talking about web design in a world where browsers disagree on implementation, standards are not always the standard, and your Sass code compiles into Boring Old CSS.

Don't rely on Susy to solve all your problems — the table-saw can't build your house for you. If you don't understand what Susy is doing, take a look at the output CSS files, dig around, and find your own path. Nothing here is sacred, just a set of tools to help make your life easier.