Skel

v2.2.1

Skel is a lightweight, low-level framework for building responsive sites and web apps. Inspired in part by its namesake (`/etc/skel`), it's designed to do just enough to make building responsively simpler, faster, and more effective, but not so much it gets in your way. Features include:

- ✓ Automatic **normalization** for consistency across multiple browsers and platforms.
- ✓ A **Breakpoint Manager** that makes responsive breakpoints accessible via JS (allowing for handy stuff like `if (skel.isActive('small')) { /* do something specific for small displays */ }`). It can also manage stylesheets and even override certain framework behaviors at a per-breakpoint level.
- ✓ A flexible CSS **grid system** that's responsive, nestable, configurable, and easy to use.
- ✓ A ton of configurability to make Skel do as little or as much as you want.
- ✓ Compatibility with all modern desktop browsers and mobile platforms (and yes, even IE8).

... all packaged in just a single 24kb file (`skel.min.js`) with **no** dependencies.

Getting Started

Simply load `skel.min.js` and call `skel.init()` to get the ball rolling:

```
<!DOCTYPE HTML>
<html>
    <head>
      <title>Untitled</title>
      <script src="skel.min.js"></script>
      <script>
        skel.init();
      </script>
    </head>
    <body>
      <div class="container">
        <h1>Hello World!</h1>
        <p>This is a test.</p>
      </div>
    </body>
</html>
```

This initializes Skel with its **default configuration**, which:

- Resets all browser styles (using the `normalize` method).
- Switches all elements over to the `border-box` box model.
- Gets the viewport ready for use with mobile devices (by way of an automatically-generated viewport `<meta>` tag).
- Sets up the `container` helper class (at `1140px` wide).
- Sets up the grid system (with `40px` gutters).

Of course, to get the most out of Skel you'll probably want to change/add to this by passing your own configuration to `skel.init()`. Skel supports a number of options that affect specific tools as well as the framework itself. For example, this configuration builds on the defaults by changing the reset method, changing the container width, and setting up a few breakpoints for the Breakpoint Manager:

```
skel.init({
  reset: 'full',
  containers: '95%',
  breakpoints: {
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)'
    },
    small: {
      media: '(max-width: 768px)'
    }
  }
});
```

The following sections cover each of Skel's components in detail along with any options that might affect them. A full list of options can also be found in the Configuration Reference.

Normalization

Once initialized, Skel will automatically "normalize" the browser environment to make it a more consistent and predictable place to build stuff. It does this in a few ways:

Box Model

By default, browsers start out most (if not all) elements with the `content-box` box model. However, as `border-box` is actually a much better way to go, Skel automatically applies it to **all** elements right off the bat (including the `:before` and `:after` pseudoelements).

Browser Styles

Browsers always apply a default stylesheet to pages to ensure common elements (like headings and paragraphs) have at least *some* basic styling going in. Since the exact implementation of these defaults can actually vary from browser to browser (resulting in some unpredictable effects on your own styling), Skel "resets" them so they look and work the same across the board. The extent to which this happens is determined by the `reset` option, which can be any of the following:

`"normalize"`

(Default) Resets browser styles with the Normalize.css method, which irons out browser inconsistencies but leaves basic styling intact. Best all-around approach.

`"full"`

Nukes **all** browser styles with Eric Meyer's CSS reset method. This is generally overkill, but can be useful if you're planning to style everything from scratch and don't want *any* defaults getting in your way.

`false`

Prevents Skel from resetting browser styles.

Viewport

"Mobile" browsers (like those found on iOS, Android, and Windows Phone) use the viewport `<meta>` tag to determine how a page should be sized and scaled. Since this is actually required for responsive pages to play nice on these browsers, Skel will automatically generate one for you. The content of this generated tag can be configured using the `viewport` option, which itself supports the following options:

`height`

Sets the height of the browser viewport. This can be an integer value (for example, `1280`), or `"device-height"` (the default).

`width`

Sets the width of the browser viewport. This can be an integer value (for example, `1280`), or `"device-width"` (the default).

`scalable`

Determines if users can manually zoom in/out of the page. Setting this to `true` (the default) enables user scaling, while `false` disables it.

Breakpoint Manager

The concept of a **breakpoint** is the foundation of responsive web design. Breakpoints are used to tie blocks of styling (and even entire stylesheets) to a set of viewport conditions that must be satisfied before they're applied to a page. Used effectively, a single page can be designed to work (and work well) across a myriad of display sizes.

Breakpoints are defined by CSS3 media queries, which can either go directly in your CSS (using the `@media` directive), or in your stylesheet `<link>` tags (using the `media` attribute). The Breakpoint Manager builds on this by letting you also define a copy of them in JS, for example:

HTML

```
<link rel="stylesheet" href="style.css" />
```

```
<link rel="stylesheet" href="style-medium.css" media="(min-width: 769px) and
(max-width: 1024px)" />
<link rel="stylesheet" href="style-small.css" media="(max-width: 768px)" />
```

JS

```
skel.init({
  breakpoints: {
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)'
    },
    small: {
      media: '(max-width: 768px)'
    }
  }
});
```

Doing this lets the Breakpoint Manager actively monitor your breakpoints right alongside the browser, opening the door to some very useful responsive functionality, such as:

Using a simple JS call to determine if a breakpoint is active:

```
if (skel.isActive('small')) {
  /* Do something special for small displays */
}
```

Performing actions when breakpoints activate or deactivate:

```
skel.on('+small', function() {
  /* Turn on feature for small displays */
});

skel.on('-small', function() {
  /* Turn off feature for small displays */
});
```

Applying different framework options when a particular breakpoint is active:

```
skel.init({
  containers: 1140,
  breakpoints: {
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)',
      containers: '90%'
    },
    small: {
      media: '(max-width: 768px)',
      containers: '95%'
    }
  }
});
```

Consolidating breakpoints in one place by letting the Breakpoint Manager load/unload stylesheets for you (more on this here):

```
skel.init({
  breakpoints: {
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)',
      href: 'style-medium.css'
    },
    small: {
      media: '(max-width: 768px)',
      href: 'style-small.css'
    }
  }
});
```

Setting up breakpoints

Breakpoints are configured with the `breakpoints` option, which is simply a list of key/value pairs (one for each breakpoint) set up like this:

```
skel.init({
  breakpoints: {
    breakpointName: {
      media: '...',
      option: value,
      option: value,
      option: value,
      ...
    },
    breakpointName: {
      media: '...',
      option: value,
      option: value,
      option: value,
      ...
    },
    ...
  }
});
```

You can define as many or as few breakpoints as you like. For example, this sets up three breakpoints (`large`, `medium`, and `small`):

```
skel.init({
  breakpoints: {
    large: {
      media: '(min-width: 1025px) and (max-width: 1280px)'
    },
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)'
    },
    small: {
      media: '(max-width: 768px)'
    }
  }
});
```

A breakpoint is considered "active" when the conditions of its media query are met, so in this case:

- `large` will only be active when the viewport is >= 1025px and <= 1280px.
- `medium` will only be active when the viewport is >= 769px and <= 1024px.
- `small` will only be active when the viewport is <= 768px.

More than one breakpoint can be active at a time (in the event you have media queries that overlap, which is totally fine). You can also omit a breakpoint's `media` option to make it *always* active.

Using breakpoints

Overriding Framework Options

Breakpoints can override parts of Skel's configuration when they're active. For example, the following has each breakpoint override `containers` with different values:

```
skel.init({
  containers: 1140,
  breakpoints: {
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)',
      containers: '90%'
    },
    small: {
      media: '(max-width: 768px)',
      containers: '95%'
    }
  }
});
```

This results in `containers` being:

- `1140px` when neither `medium` nor `small` is active.
- `90%` when `medium` is active.
- `95%` when `small` is active.

Breakpoints can override any of the following options:

grid*

Configures the grid system.

viewport*

Configures the viewport `<meta>` tag.

containers

Sets the width of the `container` helper class.

**Individual sub-options can also be overriden.*

Note that if more than one breakpoint is active, overrides will be **inherited** based on breakpoint order (ie. where they show up in your breakpoint list). For example, let's say you add a new `xsmall` breakpoint that's active whenever the viewport is <= 480px:

```
skel.init({
  containers: 1200,
  breakpoints: {
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)',
      containers: '90%'
    },
    small: {
      media: '(max-width: 768px)',
      containers: '95%'
    },
    xsmall: {
      media: '(max-width: 480px)'
    }
  }
});
```

While xsmall doesn't explicitly override containers, it'll inherit the override set by small because a) small will always be active whenever xsmall is (because <= 480px <= 768px), and b) small is defined in the breakpoint list *before* xsmall. As a result, whenever xsmall is active, containers will be the value set by small – 95%. However, if you decide to have xsmall explicitly override containers:

```
skel.init({
  containers: 1200,
  breakpoints: {
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)',
      containers: '90%'
    },
    small: {
      media: '(max-width: 768px)',
      containers: '95%'
    },
    xsmall: {
      media: '(max-width: 480px)',
      containers: '100%'
    }
  }
});
```

Now xsmall's override takes precedence, resulting in containers being 100% whenever it's active, and 95% when just small is active.

API

The Breakpoint Manager exposes a handful of API functions via the skel object, a full list of which can be found here.

Media Query Consolidation

One of the drawbacks of defining your breakpoints in JS is the fact that you still have to define them elsewhere (which is very non-DRY). For example:

```html
<!DOCTYPE HTML>
<html>
  <head>
    <title>Untitled</title>
    <link rel="stylesheet" href="style.css" />
    <link rel="stylesheet" href="style-medium.css" media="(min-width: 769px) and
(max-width: 1024px)" />
    <link rel="stylesheet" href="style-small.css" media="(max-width: 768px)" />
    <script src="skel.min.js"></script>
    <script>
      skel.init({
        breakpoints: {
          medium: {
            media: '(min-width: 769px) and (max-width: 1024px)'
          },
          small: {
            media: '(max-width: 768px)'
          }
        }
      });
    </script>
  </head>
  <body>
    <div class="container">
      <h1>Hello World!</h1>
      <p>This is a test.</p>
    </div>
  </body>
</html>
```

However, in situations where whole stylesheets are being tied to breakpoints (as in the above example), the Breakpoint Manager can actually handle loading/unloading them for you by simply setting the `href` option on each of your breakpoints:

```html
<!DOCTYPE HTML>
<html>
  <head>
    <title>Untitled</title>
    <link rel="stylesheet" href="style.css" />
    <script src="skel.min.js"></script>
    <script>
      skel.init({
        breakpoints: {
          medium: {
            media: '(min-width: 769px) and (max-width: 1024px)'
            href: 'style-medium.css'
          },
          small: {
            media: '(max-width: 768px)'
            href: 'style-small.css'
          }
        }
      });
    </script>
  </head>
  <body>
    <div class="container">
      <h1>Hello World!</h1>
      <p>This is a test.</p>
    </div>
  </body>
</html>
```

This allows you to eliminate some (or in some cases all) of your `<link>` tags while neatly consolidating all media queries in one place. A few things to be aware of when using this approach:

- Clients with JS disabled won't see any of the stylesheets handled by the Breakpoint Manager, but this can be overcome using a `<noscript>` block that loads `skel.css` (which contains all of Skel's built-in CSS) and just the stylesheets you want applied in those situations.
- In some instances, pages with many external scripts may experience what's known as FOUC (a Flash Of Unstyled Content) when the page loads for the first time. However, changing the load order of your scripts and/or the order in which stuff gets initialized is usually enough to fix this.
- This has the positive side effect of allowing the use of simple ranged media queries (like `(max-width: 768px)` or `(min-width: 769px) and (max-width: 1024px)`) on older browsers such as IE8.

Helper Classes

Skel automatically makes the following "helper classes" available once it's initialized:

Container

The `container` class is used to create a general purpose container element, which is simply an element with a defined (and usually fixed) width that's horizontally centered. For example:

HTML

```
<div class="container">
  <h2>A Container</h2>
  <p>This is a container. It's typically used to give content
  fixed boundaries so it doesn't look all weird and stretched
  out on large displays.</p>
</div>
```

The width of all containers is determined by the `containers` option (which can be any valid CSS measurement, such as `960`, `'960px'`, `'60em'`, `'75%'`, or `'30vw'`). If you're using the Breakpoint Manager, you can also set `containers` on your breakpoints (resulting in different container widths depending on which of them are active).

Modifiers

Individual container elements can also be paired with one of the following modifier classes (applied to the container's class attribute, eg. `<div class="container 75%">`):

`25%`

Narrows this container to 25% normal container width.

`50%`

Narrows this container to 50% normal container width.

`75%`

Narrows this container to 75% normal container width.

`125%`

Widens this container to 125% normal container width.

For example:

```
<div class="container 125%">
  This container is 125% normal width.
</div>
<div class="container">
  This is a normal container.
</div>
<div class="container 75%">
```

```
  This container is 75% normal width.
</div>
```

Locking

By appending `!` to the value of `containers`, **all** containers will be "locked" to that value regardless of any modifiers they may be using. For example, the following locks all containers to `95%`:

```
containers: "95%!"
```

This comes in handy when creating breakpoints that target narrower displays (where containers using, say, the `25%` modifier may look a bit weird). For example, this locks all containers to `95%` when `small` is active:

```
skel.init({
  containers: 1200,
  breakpoints: {
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)',
      containers: '90%'
    },
    small: {
      media: '(max-width: 768px)',
      containers: '95%!'
    },
  }
});
```

Conditionals

Note: Only available when using the Breakpoint Manager.

Skel generates **conditional classes** that can be used to hide/show elements based on whether a given breakpoint is active or inactive. These take the form of:

`only-breakpointName`

Only show this element when `breakpointName` is active.

`not-breakpointName`

> Only show this element when `breakpointName` is **not** active.

For example, given these breakpoints:

```
skel.init({
  breakpoints: {
    large: {
      media: '(min-width: 1025px) and (max-width: 1280px)'
    },
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)'
    },
    small: {
      media: '(max-width: 768px)'
    }
  }
});
```

Skel will automatically generate the following classes:

`only-large`

> Visible only when `large` is active.

`not-large`

> Hidden when `large` is active.

`only-medium`

> Visible only when `medium` is active.

`not-medium`

> Hidden when `medium` is active.

`only-small`

> Visible only when `small` is active.

`not-small`

> Hidden when `small` is active.

You can then use these classes to do stuff like:

```
<p class="only-large">You can only see this on large displays.</p>
<p class="only-medium">You can only see this on medium displays.</p>
<p class="only-small">You can only see this on small displays.</p>
<p class="not-large">You can see this on anything but a large display.</p>
<p class="not-medium">You can see this on anything but a medium display.</p>
<p class="not-small">You can see this on anything but a small display.</p>
```

Breakpoints

Note: Only available when using the Breakpoint Manager.

The names of all active breakpoints are automatically appended to the `<html>` element's `class` attribute (in the order they show up in the breakpoint list). For example, given these breakpoints:

```
skel.init({
  breakpoints: {
    large: {
      media: '(min-width: 1025px) and (max-width: 1280px)'
    },
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)'
    },
    small: {
      media: '(max-width: 768px)'
    },
    xsmall: {
      media: '(max-width: 480px)'
    }
  }
});
```

You'll get this when `medium` is active:

```
<html class="medium">
  ...
</html>
```

And this when `xsmall` is active:

```
<html class="small xsmall">
```

```
  ...
</html>
```

These can be used in a variety of ways. For example, limiting styling to a specific breakpoint:

```
#header {
  /* Regular header styling */
}

html.small #header {
  /* Special header styling for small displays */
}
```

Grid System

One of Skel's most important tools is its **grid system**, which provides a simple, structured way to quickly build out responsive page layouts. Grids are flexible, nestable, fully responsive, and incredibly easy to create. They're made up of two components:

Cells

- Where content lives.
- Assigned a **unit width** (by way of a class) to indicate how much space it takes up when placed in a row (see below).
- Can be anywhere from 1 unit wide (`1u`) to 12 units wide (`12u`).

Here's what 6 unit-wide (`6u`) cell looks like:

```
<div class="6u">
  <h2>Hi!</h2>
  <p>I'm a 6 unit-wide cell.</p>
</div>
```

Rows

- Where cells live.

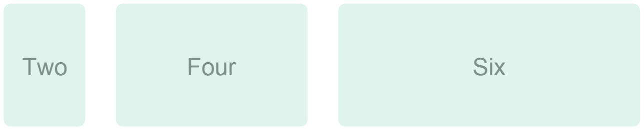
- Can hold any number of cells in any order.
- Contained cells automatically wrap to new lines every 12 units *or* immediately following a cell that includes a **terminator** (`$`) after its unit width (eg. `6u$`).
- Contained cells are spaced out with **vertical gutters**.
- Adjacent rows and lines *within* rows are (optionally) spaced out with **horizontal gutters**.
- Fluid width, so they'll expand or contract to fill whatever space is available (proportionally resizing all contained cells in the process).

Here's what a row with three cells (a `2u`, a `4u`, and a `6u`) looks like:

HTML

```
<div class="row">
  <div class="2u">Two</div>
  <div class="4u">Four</div>
  <div class="6u">Six</div>
</div>
```

Output

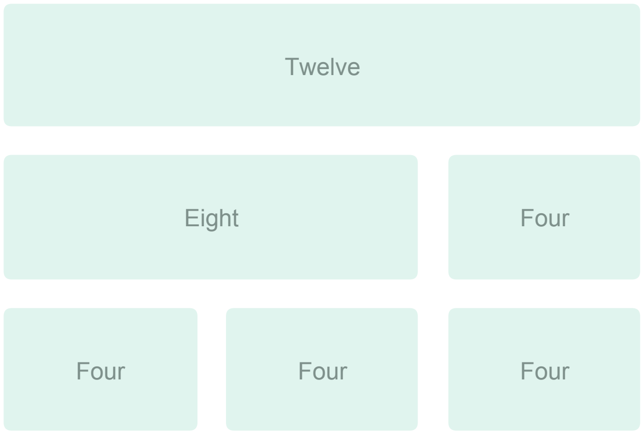


You only need a single row to create a grid, but you can combine them to create more complex layouts. For example:

HTML

```
<div class="row">
  <div class="12u">Twelve</div>
</div>
<div class="row">
  <div class="8u">Eight</div>
  <div class="4u">Four</div>
</div>
<div class="row">
  <div class="4u">Four</div>
  <div class="4u">Four</div>
  <div class="4u">Four</div>
</div>
```

Output

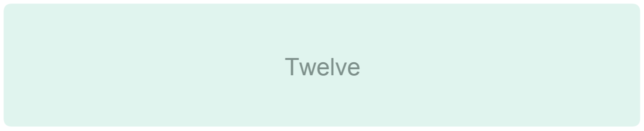


You can also create the same layout as above using just a single row and terminators (placed every 12 units):

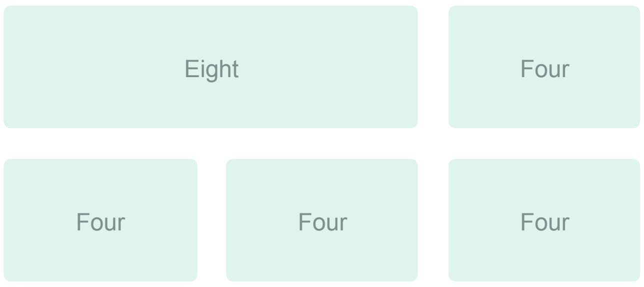
HTML

```
<div class="row">
  <div class="12u$">Twelve</div>
  <div class="8u">Eight</div>
  <div class="4u$">Four</div>
  <div class="4u">Four</div>
```

Output



```
  <div class="4u">Four</div>
  <div class="4u$">Four</div>
</div>
```

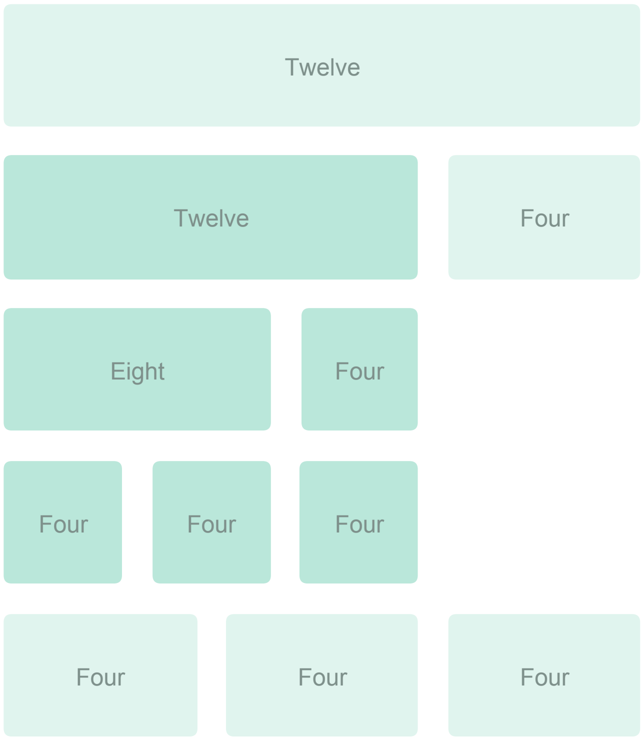


You can even **nest** rows *inside* your cells:

HTML

```
<div class="row">
  <div class="12u">Twelve</div>
</div>
<div class="row">
  <div class="8u">
    <div class="row">
      <div class="12u">Twelve</div>
    </div>
    <div class="row">
      <div class="8u">Eight</div>
      <div class="4u">Four</div>
    </div>
    <div class="row">
      <div class="4u">Four</div>
      <div class="4u">Four</div>
      <div class="4u">Four</div>
    </div>
  </div>
  <div class="4u">Four</div>
</div>
<div class="row">
  <div class="4u">Four</div>
  <div class="4u">Four</div>
  <div class="4u">Four</div>
</div>
```

Output

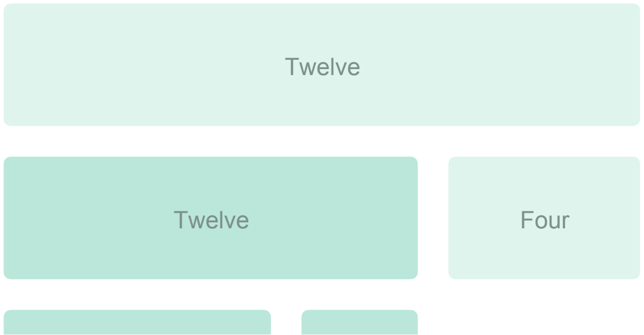


Which works just as well when using terminators:

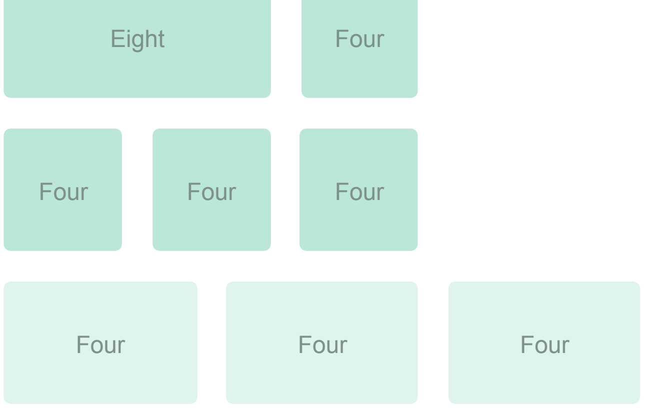
HTML

```
<div class="row">
  <div class="12u$">Twelve</div>
  <div class="8u">
    <div class="row">
      <div class="12u$">Twelve</div>
      <div class="8u">Eight</div>
      <div class="4u$">Four</div>
```

Output



```
      <div class="4u">Four</div>
      <div class="4u">Four</div>
      <div class="4u$">Four</div>
    </div>
  </div>
  <div class="4u$">Four</div>
  <div class="4u">Four</div>
  <div class="4u">Four</div>
  <div class="4u$">Four</div>
</div>
```



Some aspects of the grid system can be configured by way of the `grid` option (which, if you're using the Breakpoint Manager, can also be applied to specific breakpoints).

Gutters

Gutters are the gaps placed between cells. The grid system uses `40px` **vertical gutters** by default, but this can be changed to any valid CSS measurement using the `gutters` grid option:

```
grid: {
  gutters: [vertical, horizontal]
}
```

For example, this changes your vertical gutters to `1.5em`:

```
grid: {
  gutters: ['1.5em', 0]
}
```

The grid system also supports **horizontal gutters** (set to `0` by default) to space out adajcent rows and lines of cells within rows. For example, to use `60px` vertical gutters and `60px` horizontal gutters:

```
grid: {
  gutters: [60, 60]
}
```

Note that if you want to use the same value for both types of gutters (as in the above example), the following shorthand will also do the trick:

```
grid: {
  gutters: 60
}
```

Row Modifiers

Rows support the following modifier classes (applied to the row's class attribute, eg. `<div class="row 200%">`):

`0%`

Removes gutters from this row's cells.

`25%`

Decreases gutters for this row's cells to 25% normal size.

`50%`

Decreases gutters for this row's cells to 50% normal size.

`150%`

Increases gutters for this row's cells to 150% normal size.

`200%`

Increases gutters for this row's cells to 200% normal size.

`uniform`

Applies horizontal gutters to this row's cells that are **equal** to its vertical gutters.

`no-collapse` (deprecated)

Prevents this row from collapsing.

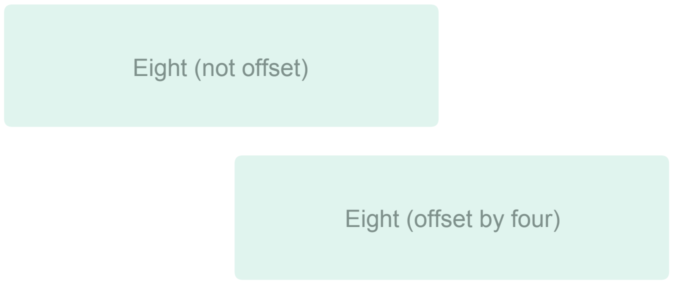
Offsetting

Cells can be offset (basically "nudged") by a number of units using an **offset class**. Offset classes take the form of `-Nu` (where `N` is the number of units to offset). For example:

HTML

```
<div class="row">
  <div class"8u">
    Eight (not offset)
  </div>
</div>
<div class="row">
  <div class="8u -4u">
    Eight (offset by four)
  </div>
</div>
```

Output

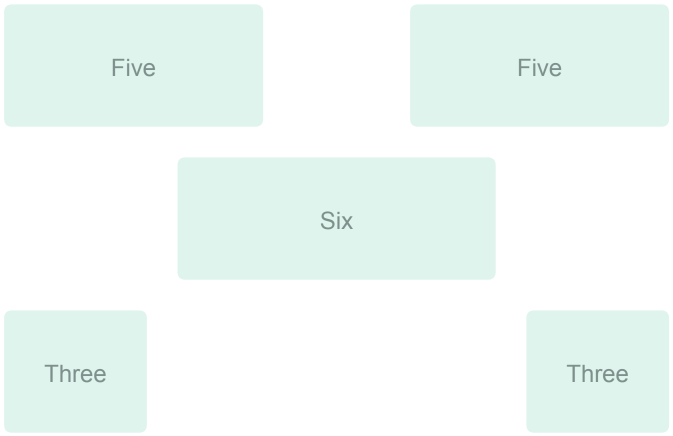


You can also offset multiple cells within a row:

HTML

```
<div class="row">
  <div class="5u">
    Five
  </div>
  <div class="5u -2u">
    Five
  </div>
</div>
<div class="row">
  <div class="6u -3u">
    Six
  </div>
</div>
<div class="row">
  <div class="3u">
    Three
  </div>
  <div class="3u -6u">
    Three
  </div>
</div>
```

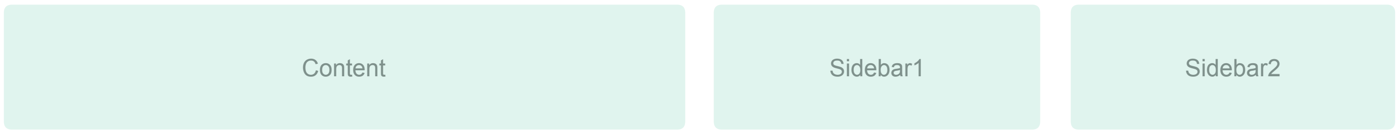
Output



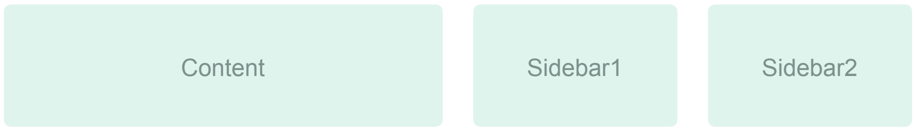
Note: Offsets take up row space just like any other cell. They can also be used responsively (for example, `4u -4u 5u(medium) -1u(medium)`).

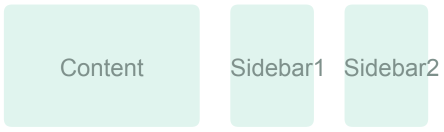
Responsiveness

Occasionally, you may run into a situation where a grid layout doesn't really play well responsively. For example, this grid works fine on a large display:

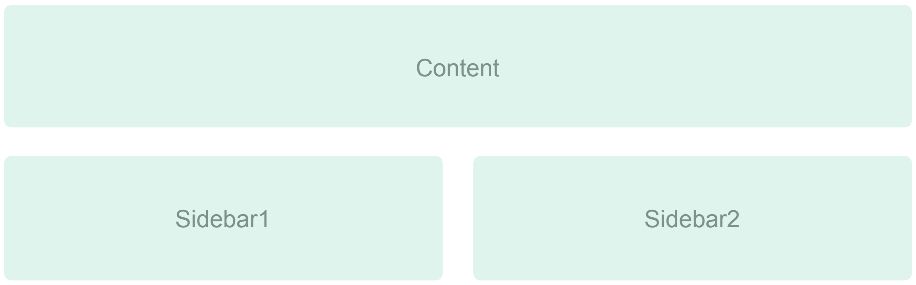


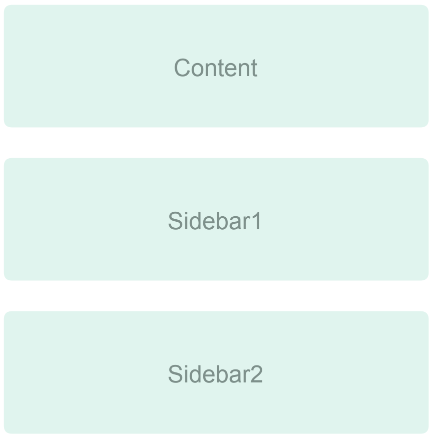
But clearly, not so well on narrower ones:





In which case, being able to rearrange it into something more optimal (like the layouts below) would be incredibly convenient:





When used in conjunction with the Breakpoint Manager, the grid system gives *all* grids, no matter how simple or complex, the ability to do just that.

How It Works

First, set up your breakpoints:

```
skel.init({
  containers: 1200,
  breakpoints: {
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)',
      containers: '90%'
    },
    small: {
      media: '(max-width: 768px)',
      containers: '95%'
    },
    xsmall: {
      media: '(max-width: 480px)'
    }
  }
});
```

Now, simply assign your grid's cells one or more **alternate unit widths** in the format of `Nu(breakpointName)` (or `Nu$(breakpointName)` when using a terminator) where `N` is the width and `breakpointName` is the breakpoint where it kicks in. For example:

```
<div class="3u 6u(small) 12u(xsmall)">
  Sidebar1
</div>
```

This cell will now automatically switch between three different widths (`3u`, `6u`, or `12u`) depending on which breakpoints are active, specifically:

- `3u` when neither `small` nor `xsmall` are active.
- `6u` when `small` is active.
- `12u` when `xsmall` is active.

Note: Precedence is determined by the order of your configuration's breakpoint list (lower in the list = higher in precedence). In this case, if `medium`, `small`, and `xsmall` are active at the same time, `xsmall` takes precedence along with its alternate width (`12u`) because it's defined after both `medium` and `small`.

A Basic Example: That Grid From Earlier

With alternate unit widths assigned to multiple cells, a grid can very easily rearrange itself into a number of different (sometimes *very* different) layouts. For example, here's a modified version of that grid from earlier:

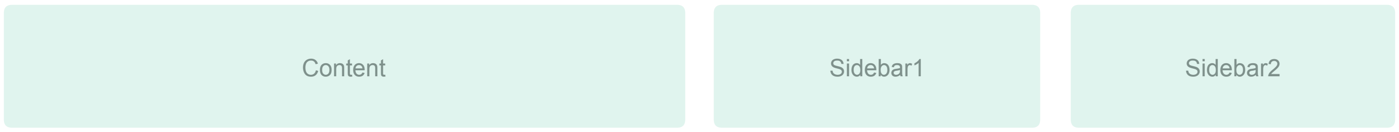
Configuration

```
skel.init({
  containers: 1200,
  breakpoints: {
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)',
      containers: '90%'
    },
    small: {
      media: '(max-width: 768px)',
      containers: '95%'
    },
    xsmall: {
      media: '(max-width: 480px)'
    }
  }
});
```

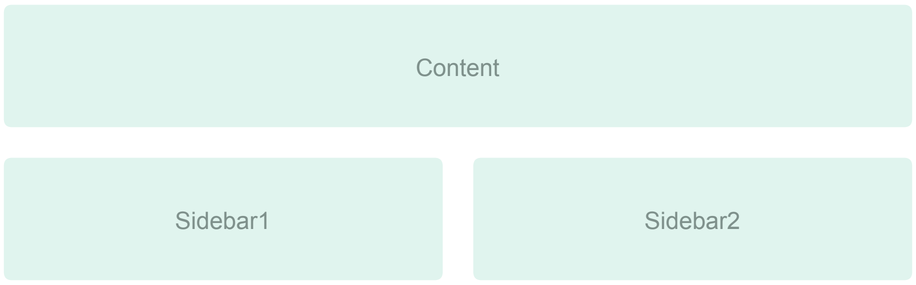
HTML

```
<div class="row">
  <div class="6u 12u$(small)">
    Content
  </div>
  <div class="3u 6u(small) 12u$(xsmall)">
    Sidebar1
  </div>
  <div class="3u$ 6u$(small) 12u$(xsmall)">
    Sidebar2
  </div>
</div>
```

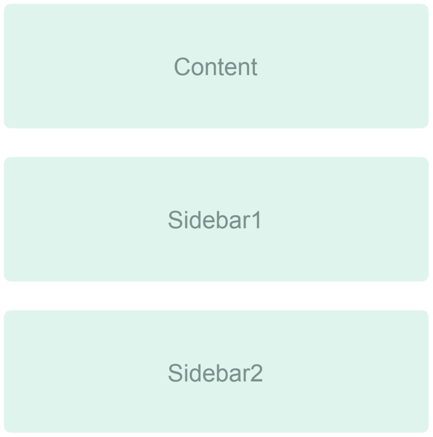
Output when neither `small` nor `xsmall` are active



Output when `small` is active



Output when `xsmall` is active



An Advanced Example

Of course, this feature really shines when it's used with more complex layouts, for instance:

```
<div class="row">
  <div class="3u 6u(medium) 12u$(xsmall)">
    <h3>Feature 1</h3>
    <p>Nunc lacinia ante nunc ac lobortis. Interdum adipiscing aliquet
    viverra nibh in adipiscing blandit tempus accumsan.</p>
  </div>
  <div class="3u 6u$(medium) 12u$(xsmall)">
    <h3>Feature 2</h3>
    <p>Nunc lacinia ante nunc ac lobortis. Interdum adipiscing aliquet
    viverra nibh in adipiscing blandit tempus accumsan.</p>
  </div>
  <div class="3u 6u(medium) 12u$(xsmall)">
    <h3>Feature 3</h3>
    <p>Nunc lacinia ante nunc ac lobortis. Interdum adipiscing aliquet
    viverra nibh in adipiscing blandit tempus accumsan.</p>
  </div>
  <div class="3u$ 6u$(medium) 12u$(xsmall)">
    <h3>Feature 4</h3>
    <p>Nunc lacinia ante nunc ac lobortis. Interdum adipiscing aliquet
    viverra nibh in adipiscing blandit tempus accumsan.</p>
  </div>
</div>
<div class="row uniform 50%">
  <div class="2u 3u(medium) 4u(small) 6u(xxsmall)"><span
class="thumbnail">1</span></div>
  <div class="2u 3u(medium) 4u(small) 6u(xxsmall)"><span
class="thumbnail">2</span></div>
  <div class="2u 3u(medium) 4u(small) 6u(xxsmall)"><span
class="thumbnail">3</span></div>
  <div class="2u 3u(medium) 4u(small) 6u(xxsmall)"><span
class="thumbnail">4</span></div>
```

```
  <div class="2u 3u(medium) 4u(small) 6u(xxsmall)"><span
class="thumbnail">5</span></div>
  <div class="2u 3u(medium) 4u(small) 6u(xxsmall)"><span
class="thumbnail">6</span></div>
</div>
```

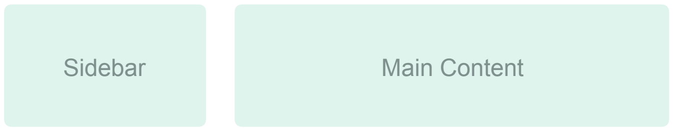
Prioritizing a Cell

When using this feature, you may run into a situation where, at a certain breakpoint, an important cell (say, your main content) gets pushed *below* a less important one (like, say, a sidebar). For example:

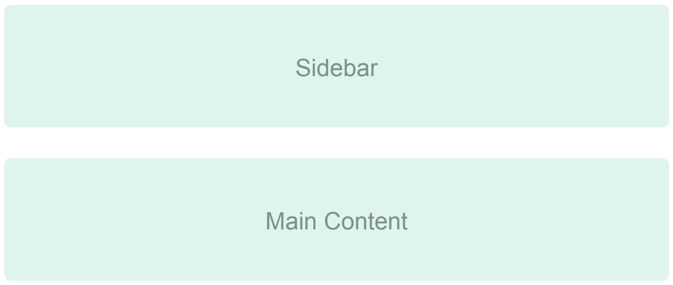
HTML

```
<div class="row">
  <div class="4u 12u$(small)">
    Sidebar
  </div>
  <div class="8u$ 12u$(small)">
    Main Content
  </div>
</div>
```

Output (default)



Output (when `small` is active)

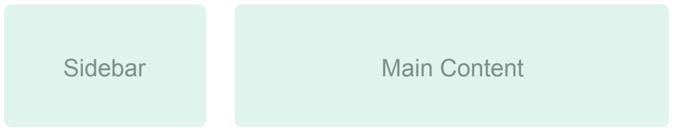


... which is probably not what you want. To prevent this, give your important cell the `important(breakpointName)` class to temporarily move it to the front of its row whenever `breakpointName` is active. In this case, applying it to the main content cell with `small` pushes it back *above* the sidebar as desired:

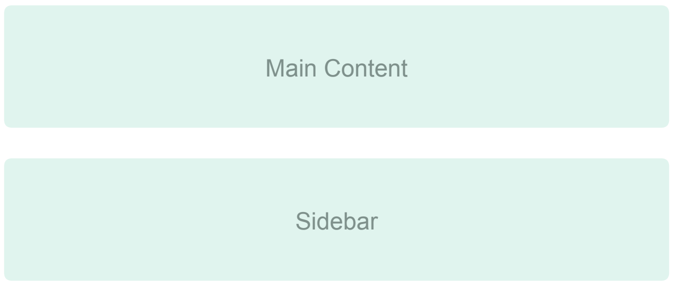
HTML

```
<div class="row">
  <div class="4u 12u$(small)">
    Sidebar
  </div>
  <div class="8u$ 12u$(small)
important(small)">
    Main Content
  </div>
</div>
```

Output (default)



Output (when `small` is active)

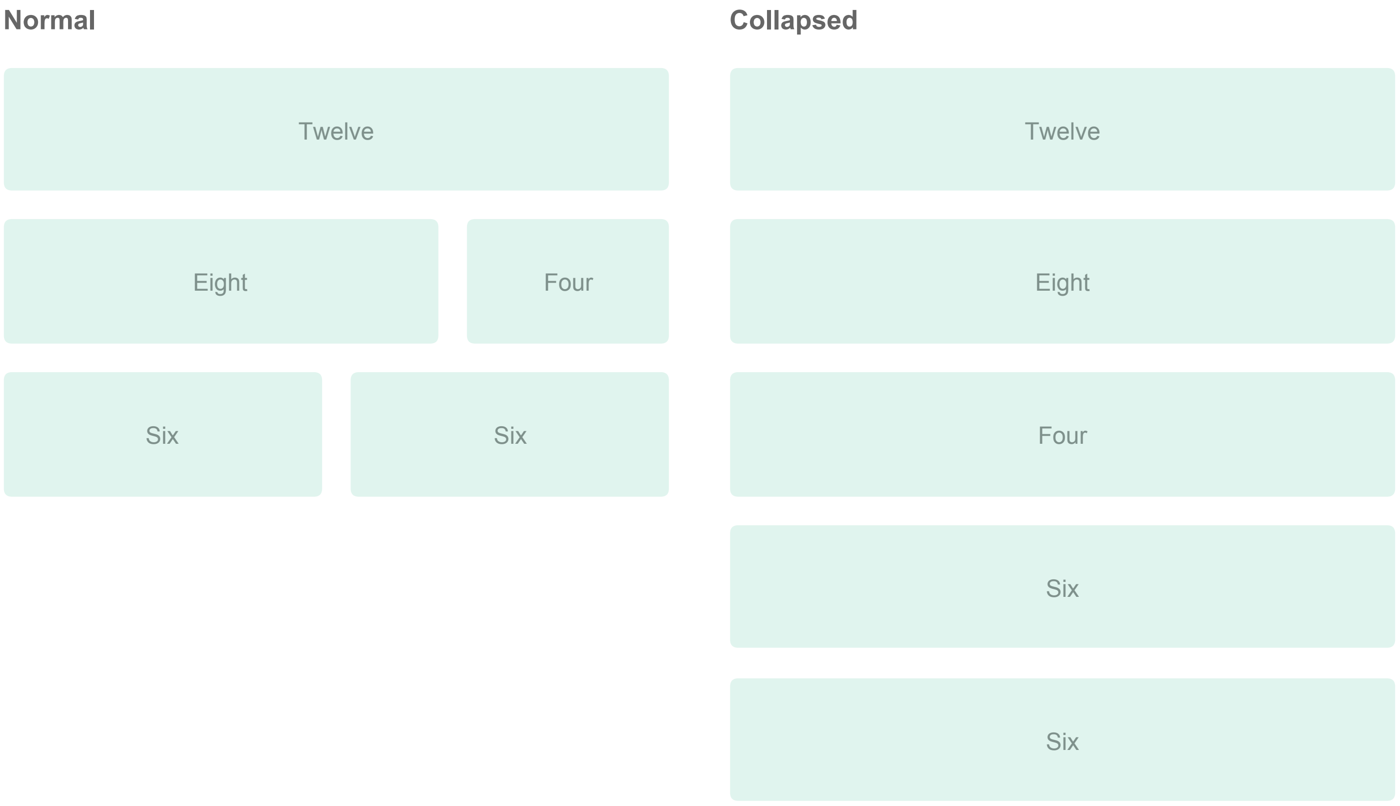


Note: The `important(breakpointName)` class can only be used once per row.

Collapsing (deprecated)

Note: Collapsing is the old (and deprecated) way to "responsify" grids. For a more elegant approach, use the grid system's built-in responsive support instead.

Instead of using the grid system's responsive support, all grids can be instructed to simply **collapse** all of their rows, forcing cells to become fluid, 100% wide, and stacked on top of each other. The end result is something that looks a whole lot better on narrower displays (like those found on mobile devices):



Collapsing can be enabled by setting the `collapse` grid option to `true`:

```
grid: {
  collapse: true
}
```

However, you'll get the most out of this feature when using it in conjunction with the Breakpoint Manager. The following, for example, tells the grid system to collapse its rows whenever the `small` breakpoint is active (and, due to the way inheritance works, also whenever `xsmall` is active):

```
skel.init({
  breakpoints: {
    large: {
      media: '(min-width: 1025px) and (max-width: 1280px)',
      containers: 960
    },
    medium: {
      media: '(min-width: 769px) and (max-width: 1024px)',
      containers: '90%'
    },
    small: {
      media: '(max-width: 768px)',
      containers: '95%',
      grid: {
        collapse: true
      }
    },
    xsmall: {
      media: '(max-width: 480px)'
    }
  }
});
```

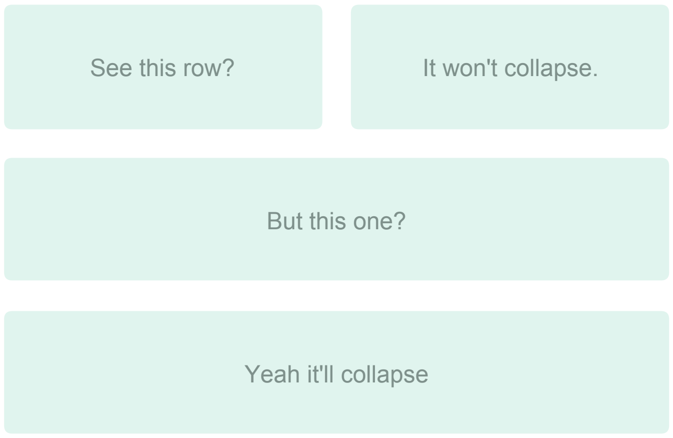
Preventing Specific Rows from Collapsing

If you have a row you *don't* want collapsed under any circumstances, give it the `no-collapse` row modifier. For example:

HTML

```
<div class="row no-collapse">
  <div class="6u">
    See this row?
  </div>
  <div class="6u">
    It won't collapse.
  </div>
</div>
<div class="row">
  <div class="6u">
    But this one?
  </div>
  <div class="6u">
    Yeah it'll collapse.
  </div>
</div>
```

Output (collapsed)



See this row? | It won't collapse.

But this one?

Yeah it'll collapse

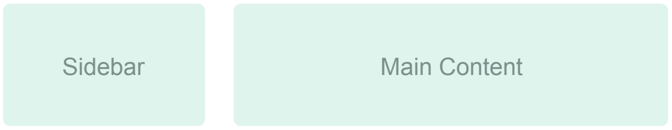
Prioritizing a Cell on Collapse

In some situations, collapsing all rows might result in cells with important content being pushed below others with less important content. For example, given a two column page layout with a sidebar on the left and the main content on the right, the latter will appear *below* the sidebar when collapse is enabled:

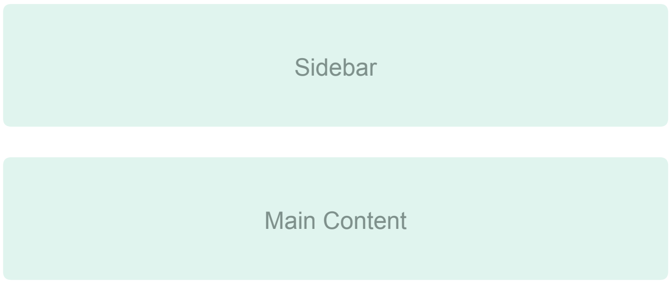
HTML

```html
<div class="row">
  <div class="4u">
    Sidebar
  </div>
  <div class="8u">
    Main Content
  </div>
</div>
```

Output



Output (collapsed)

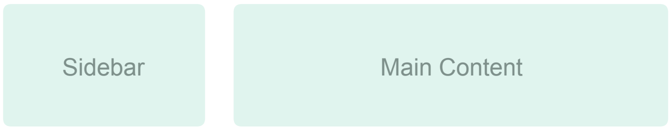


If you don't want this to happen, assign the more important cell the `important(collapse)` class to *temporarily* move it to the front of its row when collapse is enabled. In the case of the above example, applying it to the main content cell now results in it appearing *above* the sidebar:

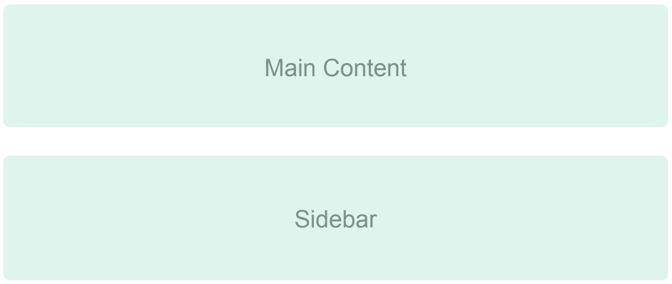
HTML

```html
<div class="row">
  <div class="4u">
    Sidebar
  </div>
  <div class="8u important(collapse)">
    Main Content
  </div>
</div>
```

Output



Output (collapsed)



Note: The `important(collapse)` class can only be used once per row.

API

Skel exposes the following methods and properties via the `skel` object:

`vars`

Object providing read-only information about Skel's status and the browser environment.

`stateId`

Current state ID. A **state**, in Breakpoint Manager terminology, is a specific combination of active breakpoints, while a **state ID** is simply the unique identifier used to keep track of that state internally. For example, given the breakpoints `medium`, `small`, and `xsmall` (defined in that exact order):

Active Breakpoints	Value of `stateId`
`medium`	`/medium`
`small`	`/small`
`small` and `xsmall`	`/small/xsmall`
(none)	`/`

While `stateId` is primarily meant for Skel's own internal use, it can be handy in other situations (for example, to perform an action only when a very specific combination of breakpoints is active).

`lastStateId`

The value of `stateId` **before** the last state change. This will be `null` if the state hasn't changed yet.

`IEVersion`

If the client is using Internet Explorer, this is set to its version number (eg. `8` for IE8, `11` for IE11). A value of `99` indicates the client isn't using IE.

`deviceType`

Client's device type. Can be:

Device Type	Value of `deviceType`
Android	`android`
iOS	`ios`
Windows Phone	`wp`
Mac OS X	`mac`
Windows	`windows`

`deviceVersion`

Client's device version.

`isTouch`

Set to `true` if the client is using a device with touch capabilities, or `false` if not.

`isMobile`

Set to `true` if the client is using a "mobile" platform (iOS, Android, or Windows Phone), or `false` if not.

`isActive(`*`breakpointName`*`)`

Determines if `breakpointName` is currently active. For example:

```
if (skel.isActive('small'))
  alert('The "small" breakpoint is active.');
else
  alert('The "small" breakpoint is NOT active.');
```

`wasActive(`*`breakpointName`*`)`

Determines if `breakpointName` *was* active before the last state change. For example:

```
if (skel.wasActive('small'))
  alert('The "small" breakpoint was active.');
else
  alert('The "small" breakpoint was NOT.');
```

`on(`*`event, handler`*`)`

Attaches a handler to one or more events. For example:

```
skel.on('+small', function() {
  /* Turn on feature for small displays */
});
```

Handlers can also be attached to multiple events by providing them in a space-delimited list:

```
skel.on('+small +medium', function() {
  /* Turn on feature for small and medium displays */
});
```

`change(`*`handler`*`)`

Attaches a handler to the `change` event.

`ready(`*`handler`*`)`

Attaches a handler to the `ready` event.

`canUseProperty(`*`property`*`)`

Determines if the browser supports a given CSS property (including those requiring a vendor prefix). For example:

```
if (skel.canUseProperty('transition'))
  alert('Browser supports CSS transitions.');
else
  alert('No support for CSS transitions. Lame.');
```

`lock(`*`width`*`[, `*`height`*`])`

Locks* the automatically-generated viewport `<meta>` tag to a fixed width (and optionally height), then reloads the page. Used in conjunction with `skel.isLocked()` and `skel.unlock()`, this can be used to create a "Switch to Desktop" button on mobile devices. For example:

```
$('#button')
  .text('Switch to ' + (skel.isLocked() ? 'Mobile' : 'Desktop'))
  .on('click', function() {

    if (skel.isLocked())
      skel.unlock();
    else
      skel.lock(1440);

  });
```

** Lock data is stored using a client-side cookie, the behavior of which can be configured.*

`unlock()`

Clears a previously set lock and reloads the page.

`isLocked()`

Determines if a lock is currently in effect.

Events

Skel also provides a set of framework-level events to which handlers can be attached (by way of the `on()` API method). The following events are currently supported:

`change`

Triggered when the Breakpoint Manager's state changes.

```
skel.on('change', function() {
  alert('Breakpoint state changed!');
});
```

`init`

Triggered when Skel initializes.

```
skel.on('init', function() {
  alert('Initialized!');
});
```

`ready`

Triggered when the DOM is ready.

```
skel.on('ready', function() {
  alert('DOM is ready!');
});
```

`+breakpointName`

Triggered when `breakpointName` activates. For example:

```
skel.on('+small', function() {
  /* Turn on feature for small displays */
});
```

`-breakpointName`

Triggered when `breakpointName` deactivates. For example:

```
skel.on('-small', function() {
  /* Turn off feature for small displays */
});
```

`!breakpointName`

Triggered when Skel first initializes and `breakpointName` is **not** active. For example:

```
skel.on('!small', function() {
  /* Turn on feature for non-small displays */
});
```

Configuration Reference

`breakpoints`

Type:	List of Breakpoint Configurations
Default:	`{}`

A list of breakpoints for the Breakpoint Manager, set up like this:

```
{
  breakpointName: { /* Breakpoint Configuration */ },
  breakpointName: { /* Breakpoint Configuration */ },
  breakpointName: { /* Breakpoint Configuration */ }
  ...
}
```

`containers`

Type:	integer, string
Default:	`1140`

Width of container elements. Can be any valid CSS measurement (eg. `960`, `'960px'`, `'30em'`, `'25vw'`).

`grid`

Type:	Grid Configuration
Default:	`{ gutters: [vertical, horizontal], collapse: false }`

Grid configuration. Each of its sub-options can be overridden at a per-breakpoint level using the `grid` breakpoint option.

`lock.path`

Type:	mixed
Default:	`false`

Sets the lock's path. A value of `false` applies the lock to just the current path, while a value of "`/`" applies it sitewide.

`lock.permanent`

Type:	bool
Default:	`true`

If `true`, makes the lock permanent until it's explicitly unlocked. If `false`, clears the lock at the end of client's session.

`pollOnce`

Type:	bool
Default:	`false`

If `true`, the Breakpoint Manager will only poll the viewport's size on initialization (as opposed to every time its size or orientation changes).

`preload`

Type:	bool
Default:	`false`

If `true`, the Breakpoint Manager will preload any stylesheets you've associated with your breakpoints (using the `href` option).

`reset`

Type:	string
Default:	`"normalize"`

Determines how Skel should reset browser styles. Can be:

`"normalize"`

Resets browser styles using the Normalize.css method.

`"full"`

Resets all browser styles using Eric Meyer's CSS reset method.

`false`

Prevents Skel from resetting browser styles.

`RTL`

Type:	bool
Default:	`false`

If `true`, Skel will make adjustments to accommodate RTL (right-to-left) languages (for example, reversing the order of cells when the grid system is collapsed).

`viewport`

Type:	Viewport Configuration
Default:	`{ width: 'device-width', scalable: true }`

Viewport configuration. Each of its sub-options can be overridden at a per-breakpoint level using the `viewport` breakpoint option.

Breakpoint Configuration

`containers`

Type:	integer, string
Default:	*undefined*

Width of container elements when this breakpoint is active. Can be any valid CSS measurement (eg. `960`, `'960px'`, `'30em'`, `'25vw'`) or undefined.

`grid`

Type:	Grid Configuration
Default:	*undefined*

Grid configuration when this breakpoint is active.

`href`

Type:	string
Default:	`false`

Associates a stylesheet with this breakpoint, which the Breakpoint Manager will then automatically load or unload as this breakpoint becomes active or inactive. As with a `<link>` tag's `href` attribute, its value can be any valid URL, eg.

```
href: "assets/css/style-small.css",
href: "/assets/css/style-small.css",
href: "http://domain.tld/assets/css/style-small.css"
```

Setting this to `false` (the default) indicates this breakpoint doesn't have a stylesheet associated with it.

`media`

Type:	string
Default:	*undefined*

Sets the CSS3 media query that must be satisfied before this breakpoint is considered active. If no media query is provided, this breakpoint will **always** be considered active.

`range` (deprecated)

Type:	string
Default:	*undefined*

Optionally used in place of `media`. Defines a simple ranged media query using the following shorthand values:

`"-X"`

Equivalent to `"(max-width: Xpx)"`

`"X-"`

Equivalent to `"(min-width: Xpx)"`

`"X-Y"`

Equivalent to `"(min-width: Xpx) and (max-width: Ypx)"`

For example:

```
range: "-768"
```

is equivalent to:

```
media: "(max-width: 768px)"
```

`viewport`

Type:	Viewport Configuration
Default:	*undefined*

Viewport configuration when this breakpoint is active.

Grid Configuration

`collapse` (deprecated)

Type:	bool
Default:	`false`

If `true`, collapses all rows (except for those with the `no-collapse` modifier). Typically used in conjunction with the Breakpoint Manager and breakpoints targeting smaller viewports (like mobile devices).

`gutters`

Type:	mixed

Default:	*undefined*

Sets the size of the gutters between cells (vertical), and between rows/lines within a row (horizontal). Can be provided in the following formats:

`[X, Y]`

Sets the vertical gutters to `X`, and horizontal gutters to `Y`.

`X`

Sets both vertical and horizontal gutters to `X`.

Where `X` and `Y` can be any valid CSS measurement (eg. `20`, `"20px"`, `"1.5em"`, `"2vw"`).

Viewport Configuration

`height`

Type:	integer, string
Default:	`"device-height"`

Sets the height of the viewport. By default this is set to `"device-height"` to lock the viewport to the height of the device (factoring in orientation). However, this can also be set to a specific pixel value (eg. `1280`).

`width`

Type:	integer, string
Default:	`"device-width"`

Sets the width of the viewport. By default this is set to `"device-width"` to lock the viewport to the width of the device (factoring in orientation). However, this can also be set to a specific pixel value (eg. `1280`).

`scalable`

Type:	bool

Default:	`true`

If `true`, users will be able to manually scale the viewport (usually with a pinch/zoom gesture). Setting this to `false` disables user scaling.

Upgrading from v1 to v2

If you're upgrading from Skel v1 to v2, here's what you need to know (and in some cases fix):

New Features

- Grid System: Rows can now hold any number of cells.
- Grid System: Added terminators.
- Grid System: Added full responsive support (deprecates collapsing).
- Helper Classes: Container widths can be locked to override modifiers.
- Helper Classes: New container modifiers (`25%` and `50%`).
- Normalization: `viewport` option now supports `height` as a sub-option.

Backwards-Incompatible Changes

- Grid System: `gutters` option syntax has been changed from an object (`{ vertical: X, horizontal: Y }`) to an array (`[X, Y]`).
- Grid System: `collapse: true` no longer forces all containers to `100%` width (now handled via container locking).
- Grid System: `skel-cell-important` renamed to `important(collapse)`.
- Helper Classes: Container modifiers renamed as follows:

v1	v2
`small`	`75%`
`large`	`125%`

- Grid System: Row modifiers renamed as follows:

v1	v2
`flush`	`0%`
`quarter`	`25%`
`half`	`50%`
`oneandhalf`	`150%`
`double`	`200%`

Credits

Skel

- CSS Resets (meyerweb.com/eric/tools/css/reset | Eric Meyer | Public domain)
- DOMReady method (github.com/ded/domready | (c) Dustin Diaz 2014 | MIT license)
- matchMedia() polyfill (github.com/paulirish/matchMedia.js | (c) 2012 Scott Jehl, Paul Irish, Nicholas Zakas, David Knight | Dual MIT/BSD license)
- Normalize (git.io/normalize | Nicolas Gallagher, Jonathan Neal | MIT License)
- UMD Wrapper (github.com/umdjs/umd | @umdjs + @nason)

License

Skel, Layers, and Baseline are released under the MIT license.