haltu / muuri

Responsive, sortable, filterable and draggable grid layouts http://haltu.github.io/muuri/

#grid #layout #sorting #drag-and-drop #filter #bin-packing

209 commits | 5 branches | 5 releases | 4 contributors | MIT

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

niklasramo v0.4.1 — Latest commit 7b1198d 5 days ago

tests	v0.4.1	5 days ago
.editorconfig	A lot of refactoring.	9 months ago
.eslintrc.js	Updated build tools.	3 months ago
.gitignore	Merge branch 'dev'	a month ago
.travis.yml	Updated unit tests.	2 months ago
AUTHORS.txt	Updated AUTHORS.txt	6 months ago
LICENSE.md	Layout system rewrite, updated demo and docs.	a year ago
README.md	v0.4.1	5 days ago
bower.json	Updated module definition logic, updated readme and added test for it…	2 months ago
gulpfile.js	Updated README and gulp config. Also made Muuri use querySelector ins…	2 months ago
karma.conf.js	Updated unit tests and added some experimental animation optimizations.	2 months ago
muuri.js	v0.4.1	5 days ago
muuri.min.js	v0.4.1	5 days ago
package-lock.json	Velocity resurrection after quick try and fail with CSS transitions.	7 days ago
package.json	v0.4.1	5 days ago

README.md

Muuri

gzip size 12.9 kB | npm v0.4.1 | cdnjs v0.4.1

Muuri creates responsive, sortable, filterable and draggable grid layouts. Yep, that's a lot of features in one library, but we have tried to make it as tiny as possible. Comparing to what's out there Muuri is a combination of Packery, Masonry, Isotope and jQuery UI sortable. Wanna see it in action? Check out the demo on the website.

Muuri's layout system allows positioning the grid items pretty much any way imaginable. The default "First Fit" bin packing layout algorithm generates similar layouts as Packery and Masonry. The implementation is heavily based on the "maxrects" approach as described by Jukka Jylänki in his research A Thousand Ways to Pack the Bin. However, you can also provide your own layout algorithm to position the items in any way you want.

Muuri uses Velocity for animating the grid items (positioning/showing/hiding) and Hammer.js for handling the dragging. And if you're wondering about the name of the library "muuri" is Finnish meaning a wall.

Table of contents

- Getting started
- API
  - Grid constructor
  - Grid options
  - Grid methods
  - Grid events

  - Item methods
- FAQ
- Credits
- License

Getting started

1. Get Muuri

Download from GitHub:

- muuri.js - for development (not minified, with comments).
- muuri.min.js - for production (minified, no comments).

Or link directly via CDNJS:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/muuri/0.4.1/muuri.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/muuri/0.4.1/muuri.min.js"></script>
```

Or install with npm:

```
npm install muuri
```

Or install with bower:

```
bower install muuri
```

2. Get the dependencies

Muuri depends on the following libraries:

- Velocity (v1.2.0+)
  - By default Muuri uses Velocity to power all the animations. However, it is possible to replace Velocity with any other animation engine by overwriting the `Muuri.ItemAnimate` constructor.
- Hammer.js (v2.0.0+)
  - Muuri uses Hammer.js to handle all the drag events. It is an optional dependency and only required if the dragging is enabled. Currently there is no easy way to use another library for handling the drag interaction. Almost all of the drag related logic exists within `Muuri.ItemDrag` constructor, which is instantiated for each item, so if you really need to customize the drag behaviour beyond what is available via the options you can replace the `Muuri.ItemDrag` constructor with your own implementation (fingers crossed).

3. Add the script tags

Add Muuri on your site and make sure to include the dependencies before Muuri.

```
<script src="velocity.js"></script>
<script src="hammer.js"></script>
<script src="muuri.js"></script>
```

4. Add the markup

- Every grid must have a container element.
- Grid items must always consist of at least two elements. The outer element is used for positioning the item and the inner element (first direct child) is used for animating the item's visibility (show/hide methods). You can insert any markup you wish inside the inner item element.

```
<div class="grid">

  <div class="item">
    <div class="item-content">
      <!-- Safe zone, enter your custom markup -->
```

```
        This can be anything.
        <!-- Safe zone ends -->
      </div>
    </div>

    <div class="item">
      <div class="item-content">
        <!-- Safe zone, enter your custom markup -->
        <div class="my-custom-content">
          Yippee!
        </div>
        <!-- Safe zone ends -->
      </div>
    </div>

  </div>
```

5. Add the styles

- The container element must be "positioned" meaning that it's CSS position property must be set to *relative*, *absolute* or *fixed*. Also note that Muuri automatically resizes the container element's width/height depending on the area the items cover and the layout algorithm configuration.
- The item elements must have their CSS position set to *absolute* and their display property set to *block*. Muuri actually enforces the `display:block;` rule and adds it as an inline style to all item elements, just in case.
- The item elements must not have any CSS transitions or animations applied to them, because they might conflict with Velocity's animations. However, the container element can have transitions applied to it if you want it to animate when it's size changes after the layout operation.
- You can control the gaps between the items by giving some margin to the item elements.

```css
.grid {
  position: relative;
}
.item {
  display: block;
  position: absolute;
  width: 100px;
  height: 100px;
  margin: 5px;
  z-index: 1;
  background: #000;
  color: #fff;
}
.item.muuri-dragging {
  z-index: 3;
}
.item.muuri-releasing {
  z-index: 2;
}
.item.muuri-hidden {
  z-index: 0;
}
.item-content {
  position: relative;
  width: 100%;
  height: 100%;
}
```

6. Fire it up

The bare minimum configuration is demonstrated below. You must always provide the container element (or a selector so Muuri can fetch the element for you), everything else is optional.

```javascript
var grid = new Muuri('.grid');
```

API

Grid constructor

`Muuri` is a constructor function and should be always instantiated with the `new` keyword. For the sake of clarity, we refer to a Muuri instance as *grid* throughout the documentation.

Syntax

`Muuri( element, [options] )`

Parameters

- **element** — *element / string*
  - Default value: `null`.
  - You can provide the element directly or use a selector (string) which uses querySelector() internally. The first element of the query's result will be used.
- **options** — *object*
  - Optional. Check out the detailed options reference.

Default options

The default options are stored in `Muuri.defaultOptions` object, which in it's default state contains the following configuration:

```
{

    // Item elements
    items: '*',

    // Default show animation
    showDuration: 300,
    showEasing: 'ease',

    // Default hide animation
    hideDuration: 300,
    hideEasing: 'ease',

    // Custom show/hide animations
    showAnimation: null,
    hideAnimation: null,

    // Item's visible/hidden state styles
    visibleStyles: {
      opacity: 1,
      scale: 1
    },
    hiddenStyles: {
      opacity: 0,
      scale: 0.5
    },

    // Layout
    layout: {
      fillGaps: false,
      horizontal: false,
      alignRight: false,
      alignBottom: false,
      rounding: true
    },
    layoutOnResize: 100,
    layoutOnInit: true,
    layoutDuration: 300,
    layoutEasing: 'ease',

    // Sorting
    sortData: null,

    // Drag & Drop
    dragEnabled: false,
    dragContainer: null,
    dragStartPredicate: {
      distance: 0,
      delay: 0,
      handle: false
    },
    dragAxis: null,
    dragSort: true,
    dragSortInterval: 100,
    dragSortPredicate: {
      threshold: 50,
      action: 'move'
    },
    dragSortGroup: null,
    dragSortWith: null,
    dragReleaseDuration: 300,
    dragReleaseEasing: 'ease',
    dragHammerSettings: {
      touchAction: 'none'
    },

    // Classnames
    containerClass: 'muuri',
    itemClass: 'muuri-item',
    itemVisibleClass: 'muuri-item-shown',
    itemHiddenClass: 'muuri-item-hidden',
    itemPositioningClass: 'muuri-item-positioning',
    itemDraggingClass: 'muuri-item-dragging',
    itemReleasingClass: 'muuri-item-releasing'

}
```

You can modify the default options easily:

```
Muuri.defaultOptions.showDuration = 400;
Muuri.defaultOptions.dragSortPredicate.action = 'swap';
```

This is how you would use the options:

```
// Minimum configuration.
var gridA = new Muuri('.grid-a');

// Providing some options.
var gridB = new Muuri('.grid-b', {
  items: '.item'
});
```

Grid options

- items
- showDuration
- showEasing
- hideDuration
- hideEasing
- visibleStyles
- hiddenStyles
- layout
- layoutOnResize
- layoutOnInit
- layoutDuration
- layoutEasing
- sortData
- dragEnabled
- dragContainer
- dragStartPredicate
- dragAxis
- dragSort
- dragSortInterval
- dragSortPredicate
- dragSortGroup
- dragSortWith
- dragReleaseDuration

- dragReleaseEasing
- containerClass
- itemClass
- itemVisibleClass
- itemHiddenClass
- itemPositioningClass
- itemDraggingClass
- itemReleasingClass

items

The initial item elements, which should be children of the container element. All elements that are not children of the container will be appended to the container. You can provide an *array* of elements, a *node list* or a selector (string). If you provide a selector Muuri uses it to filter the current child elements of the container element and sets them as initial items. By default all current child elements of the provided container element are used as initial items.

- Default value: `'*'`.
- Accepted types: array (of elements), node list, string, null.

```
// Use specific items.
var grid = new Muuri(elem, {
  items: [elemA, elemB, elemC]
});

// Use node list.
var grid = new Muuri(elem, {
  items: elem.querySelectorAll('.item')
});

// Use selector.
var grid = new Muuri(elem, {
  items: '.item'
});
```

showDuration

Show animation duration in milliseconds. Set to `0` to disable show animation.

- Default value: `300`.
- Accepted types: number.

```
var grid = new Muuri(elem, {
  showDuration: 600
});
```

showEasing

Show animation easing. Accepts any valid Velocity.js easing value.

- Default value: `'ease'`.
- Accepted types: array, string.

```
var grid = new Muuri(elem, {
  showEasing: 'ease-out'
});
```

hideDuration

Hide animation duration in milliseconds. Set to `0` to disable hide animation.

- Default value: `300`.
- Accepted types: number.

```
var grid = new Muuri(elem, {
  hideDuration: 600
});
```

hideEasing

Hide animation easing. Accepts any valid Velocity.js easing value.

- Default value: `'ease'` .
- Accepted types: array, string.

```
var grid = new Muuri(elem, {
  hideEasing: 'ease-out'
});
```

visibleStyles

The styles that will be applied to all visible items. These styles are also used for the show/hide animations which means that you have to have the same style properties in visibleStyles and hiddenStyles options.

- Default value: `{opacity: 1, scale: 1}` .
- Accepted types: object.

```
var grid = new Muuri(elem, {
  visibleStyles: {
    opacity: 1,
    rotateZ: '45deg'
  },
  hiddenStyles: {
    opacity: 0,
    rotateZ: '-45deg'
  }
});
```

hiddenStyles

The styles that will be applied to all hidden items. These styles are also used for the show/hide animations which means that you have to have the same style properties in visibleStyles and hiddenStyles options.

- Default value: `{opacity: 0, scale: 0.5}` .
- Accepted types: object.

```
var grid = new Muuri(elem, {
  visibleStyles: {
    opacity: 1,
    rotateZ: '45deg'
  },
  hiddenStyles: {
    opacity: 0,
    rotateZ: '-45deg'
  }
});
```

layout

Define how the items will be laid out. Although it's not documented well (at all) in this section, you *can* provide a function here also if you want to provide your own layout algorithm (may the source be with you).

- Default value: `{fillGaps: false, horizontal: false, alignRight: false, alignBottom: false}` .
- Accepted types: function, object.

Provide an object to configure the default layout algorithm with the following properties:

- fillGaps — *boolean*
  - Default value: `false` .

  - When `true` the algorithm goes through every item in order and places each item to the first available free slot, even if the slot happens to be visually *before* the previous element's slot. Practically this means that the items might not end up visually in order, but there will be less gaps in the grid. By default this option is `false` which basically means that the following condition will be always true when calculating the layout (assuming `alignRight` and `alignBottom` are `false`): `nextItem.top > prevItem.top || (nextItem.top === prevItem.top && nextItem.left > prevItem.left)` . This also means that the items will be visually in order.
- **horizontal** — *boolean*
  - Default value: `false` .
  - When `true` the grid works in landscape mode (grid expands to the right). Use for horizontally scrolling sites. When `false` the grid works in "portrait" mode and expands downwards.
- **alignRight** — *boolean*
  - Default value: `false` .
  - When `true` the items are aligned from right to left.
- **alignBottom** — *boolean*
  - Default value: `false` .
  - When `true` the items are aligned from the bottom up.
- **rounding** — *boolean*
  - Default value: `true` .
  - When `true` the dimensions of the items will be automatically rounded for the layout calculations using `Math.round()` . Set to `false` to use accurate dimensions. In practice you would want disable this if you are using relative dimension values for items (%, em, rem, etc.). If you have defined item dimensions with pixel values (px) it is recommended that you leave this on.

```
var grid = new Muuri(elem, {
  layout: {
    fillGaps: true,
    horizontal: true,
    alignRight: true,
    alignBottom: true,
    rounding: false
  }
});
```

layoutOnResize

Should Muuri automatically trigger `layout` method on window resize? Set to `false` to disable. When a number or `true` is provided Muuri will automatically lay out the items every time window is resized. The provided number (`true` is transformed to `0`) equals to the amount of time (in milliseconds) that is waited before items are laid out after each window resize event.

- Default value: `100` .
- Accepted types: boolean, number.

```
// No layout on resize.
var grid = new Muuri(elem, {
  layoutOnResize: false
});

// Layout on resize (instantly).
var grid = new Muuri(elem, {
  layoutOnResize: true
});

// Layout on resize (with 200ms debounce).
var grid = new Muuri(elem, {
  layoutOnResize: 200
});
```

layoutOnInit

Should Muuri trigger `layout` method automatically on init?

- Default value: `true` .
- Accepted types: boolean.

```
var grid = new Muuri(elem, {
  layoutOnInit: false
});
```

layoutDuration

The duration for item's layout animation in milliseconds. Set to `0` to disable.

- Default value: `300` .
- Accepted types: number.

```
var grid = new Muuri(elem, {
  layoutDuration: 600
});
```

layoutEasing

The easing for item's layout animation. Accepts any valid Velocity.js easing value.

- Default value: `'ease'` .
- Accepted types: string.

```
// jQuery UI easings.
var grid = new Muuri(elem, {
  layoutEasing: 'easeInSine'
});

// Custom bezier curve.
var grid = new Muuri(elem, {
  layoutEasing: [0.17, 0.67, 0.83, 0.67]
});

// Spring physics.
var grid = new Muuri(elem, {
  layoutEasing: [250, 15]
});

// Step easing.
var grid = new Muuri(elem, {
  layoutEasing: [8]
});
```

sortData

The sort data getter functions. Provide an object where the key is the name of the sortable attribute and the function returns a value (from the item) by which the items can be sorted.

- Default value: `null` .
- Accepted types: object, null.

```
var grid = new Muuri(elem, {
  sortData: {
    foo: function (item, element) {
      return parseFloat(element.getAttribute('data-foo'));
    },
    bar: function (item, element) {
      return element.getAttribute('data-bar').toUpperCase();
    }
  }
});
// Refresh sort data whenever an item's data-foo or data-bar changes
grid.refreshSortData();
// Sort the grid by foo and bar.
grid.sort('foo bar');
```

dragEnabled

Should items be draggable?

- Default value: `false` .
- Accepted types: boolean.

```
var grid = new Muuri(elem, {
  dragEnabled: true
});
```

dragContainer

The element the dragged item should be appended to for the duration of the drag. If set to `null` (which is also the default value) the grid's container element will be used.

- Default value: `null` .
- Accepted types: element, null.

```
var grid = new Muuri(elem, {
  dragContainer: document.body
});
```

dragStartPredicate

A function that determines when the item should start to move when the item is being dragged. By default uses the built-in predicate which has some configurable options.

- Default value: `{distance: 0, delay: 0, handle: false}` .
- Accepted types: function, object.

If an object is provided the default sort predicate handler will be used. You can define the following properties:

- **distance** — *number*
  - Default value: `0` .
  - How many pixels must be dragged before the dragging starts.
- **delay** — *number*
  - Default value: `0` .
  - How long (in milliseconds) the user must drag before the dragging starts.
- **handle** — *string / boolean*
  - Default value: `false` .
  - The selector(s) which much match the event target element for the dragging to start.

If you provide a function you can totally customize the drag start logic. When the user starts to drag an item this predicate function will be called until you return `true` or `false` . If you return `true` the item will begin to move whenever the item is dragged. If you return `false` the item will not be moved at all. Note that after you have returned `true` or `false` this function will not be called until the item is released and dragged again.

The predicate function receives two arguments:

- **item** — *Muuri.Item*
  - The item that's being dragged.
- **event** — *object*
  - The drag event (Hammer.js event).

```
// Configure the default preficate
var grid = new Muuri(elem, {
  dragStartPredicate: {
    distance: 10,
    delay: 100,
    handle: '.foo, .bar'
  }
});

// Provide your own predicate
var grid = new Muuri(elem, {
  dragStartPredicate: function (item, e) {
    // Start moving the item after the item has been dragged for one second.
    if (e.deltaTime > 1000) {
      return true;
    }
  }
});
```

dragAxis

Force items to be moved only vertically or horizontally when dragged. Set to `'x'` for horizontal movement and to `'y'` for vertical movement. By default items can be dragged both vertically and horizontally.

- Default value: `null`.
- Accepted types: string.
- Allowed values: `'x'`, `'y'`.

```
// Move items only horizontally when dragged.
var grid = new Muuri(elem, {
  dragAxis: 'x'
});

// Move items only vertically when dragged.
var grid = new Muuri(elem, {
  dragAxis: 'y'
});
```

dragSort

Should the items be sorted during drag?

- Default value: `true`.
- Accepted types: boolean.

```
// Disable drag sorting.
var grid = new Muuri(elem, {
  dragSort: false
});
```

dragSortInterval

Defines the amount of time the dragged item must be still before `dragSortPredicate` function is called. The default `dragSortPredicate` is pretty heavy function which means that you might see some janky animations and/or an unresponsive UI if you set this value too low (`0` is not recommended).

- Default value: `100`.
- Accepted types: number.

```
// Sort on every drag move.
var grid = new Muuri(elem, {
  dragSortInterval: 0
});

// Sort with a decent buffer.
var grid = new Muuri(elem, {
  dragSortInterval: 150
});
```

dragSortPredicate

Defines the logic for the sort procedure during dragging an item.

- Default value: `{action: 'move', tolerance: 50}`.
- Accepted types: function, object.

If an object is provided the default sort predicate handler will be used. You can define the following properties:

- **action** — *string*
  - Default value: `'move'` .
  - Allowed values: `'move'` , `'swap'` .
  - Should the dragged item be *moved* to the new position or should it *swap* places with the item it overlaps?
- **threshold** — *number*
  - Default value: `50` .
  - Allowed values: `1 - 100` .
  - How many percent the intersection area between the dragged item and the compared item should be from the maximum potential intersection area between the items before sorting is triggered.

Alternatively you can provide your own callback function where you can define your own custom sort logic. The callback function receives two arguments:

- **item** — *Muuri.Item*
  - The item that's being dragged.
- **event** — *object*
  - The drag event (Hammer.js event).

The callback should return a *falsy* value if sorting should not occur. If, however, sorting should occur the callback should return an object containing the following properties:

- **index** — *number*
  - The index where the item should be moved to.
- **grid** — *Muuri*
  - The grid where the item should be moved to.
  - Defaults to the item's current grid.
  - Optional.
- **action** — *string*
  - The movement method.
  - Default value: `'move'` .
  - Allowed values: `'move'` or `'swap'` .
  - Optional.

```
// Customize the default predicate.
var grid = new Muuri(elem, {
  dragSortPredicate: {
    threshold: 90,
    action: 'swap'
  }
});

// Provide your own predicate.
var grid = new Muuri(elem, {
  dragSortPredicate: function (item, e) {
    if (e.deltaTime > 1000) {
      return {
        index: Math.round(e.deltaTime / 1000) % 2 === 0 ? -1 : 0,
        action: 'swap'
      };
    }
  }
});
```

dragSortGroup

The grid's sort group(s), e.g. `'groupA'` or `['groupA', 'groupB']` . If you provide no sort group the grid cannot be targeted with `dragSortWith` option, which means that items can not be dragged into the grid from other grids.

- Default value: `null` .
- Accepted types: array, string, null.

```
var gridA = new Muuri(elemA, {
  dragSortGroup: 'groupA'
});
var gridB = new Muuri(elemB, {
  dragSortGroup: ['groupA', 'groupB']
});
```

dragSortWith

Defines the sort group(s) that this instance's item's can be dragged to. Provide a string to target a single sort group, e.g. `'groupA'` , or an array of targeted sort groups using an array, e.g. `['groupA', 'groupC']` .

- Default value: `null` .
- Accepted types: array, null.

```
// This grid's items can not be dragged into any other grids.
var gridA = new Muuri(elemA, {
  dragSortGroup: 'a'
});

// This grid's items can be dragged into gridC.
var gridB = new Muuri(elemB, {
  dragSortGroup: 'b',
  dragSortWith: 'c'
});

// This grid's items can be dragged into gridA and gridB.
var gridC = new Muuri(elemC, {
  dragSortGroup: 'c',
  dragSortWith: ['a', 'b']
});
```

dragReleaseDuration

The duration for item's drag release animation. Set to `0` to disable.

- Default value: `300` .
- Accepted types: number.

```
var grid = new Muuri(elem, {
  dragReleaseDuration: 600
});
```

dragReleaseEasing

The easing for item's drag release animation. Accepts any valid Velocity.js easing value.

- Default value: `'ease'` .
- Accepted types: array, string.

```
var grid = new Muuri(elem, {
  dragReleaseEasing: 'ease-out'
});
```

containerClass

Container element's classname.

- Default value: `'muuri'` .
- Accepted types: string.

```
var grid = new Muuri(elem, {
  containerClass: 'foo'
});
```

itemClass

Item element's classname.

- Default value: `'muuri-item'` .
- Accepted types: string.

```
var grid = new Muuri(elem, {
  itemClass: 'foo-item'
});
```

itemVisibleClass

Visible item's classname.

- Default value: `'muuri-item-shown'` .
- Accepted types: string.

```
var grid = new Muuri(elem, {
  itemVisibleClass: 'foo-item-shown'
});
```

itemHiddenClass

Hidden item's classname.

- Default value: `'muuri-item-hidden'` .
- Accepted types: string.

```
var grid = new Muuri(elem, {
  itemHiddenClass: 'foo-item-hidden'
});
```

itemPositioningClass

This classname will be added to the item element for the duration of positioning.

- Default value: `'muuri-item-positioning'` .
- Accepted types: string.

```
var grid = new Muuri(elem, {
  itemPositioningClass: 'foo-item-positioning'
});
```

itemDraggingClass

This classname will be added to the item element for the duration of drag.

- Default value: `'muuri-item-dragging'` .
- Accepted types: string.

```
var grid = new Muuri(elem, {
  itemDraggingClass: 'foo-item-dragging'
});
```

itemReleasingClass

This classname will be added to the item element for the duration of release.

- Default value: `'muuri-item-releasing'` .
- Accepted types: string.

```
var grid = new Muuri(elem, {
  itemReleasingClass: 'foo-item-releasing'
});
```

Grid methods

- grid.getElement()
- grid.getItems([targets], [state])
- grid.refreshItems([items])
- grid.refreshSortData([items])
- grid.synchronize()
- grid.layout([instant], [callback])
- grid.add(elements, [options])
- grid.remove(items, [options])
- grid.show(items, [options])
- grid.hide(items, [options])
- grid.filter(predicate, [options])
- grid.sort(comparer, [options])
- grid.move(item, position, [options])
- grid.send(item, grid, position, [options])
- grid.on(event, listener)
- grid.once(event, listener)
- grid.off(event, listener)
- grid.destroy([removeElements])

grid.getElement()

Get the instance element.

Returns — *element*

```
var elem = grid.getElement();
```

grid.getItems([targets], [state])

Get all items in the grid. Optionally you can provide specific targets (indices or elements) and filter the results by the items' state.

Parameters

- **targets** — *array / element / Muuri.Item / number*
  - An array of item instances/elements/indices.
  - Optional.
- **state** — *string*
  - Accepted values: `'active'`, `'inactive'`, `'visible'`, `'hidden'`, `'showing'`, `'hiding'`, `'positioning'`, `'dragging'`, `'releasing'`, `'migrating'`.
  - Default value: `undefined`.
  - Optional.

Returns — *array*

Returns the queried items.

```
// Get all items, both active and inactive.
var allItems = grid.getItems();

// Get all active (visible) items.
var activeItems = grid.getItems('active');

// Get all inactive (hidden) items.
var inactiveItems = grid.getItems('inactive');

// Get the first item (active or inactive).
var firstItem = grid.getItems(0)[0];

// Get specific items by their elements (inactive or active).
var items = grid.getItems([elemA, elemB]);
```

```
// Get specific inactive items.
var items = grid.getItems([elemA, elemB], 'inactive');
```

grid.refreshItems([items])

Refresh the cached dimensions of the grid's items. When called without any arguments all active items are refreshed. Optionally you can provide specific the items which you want to refresh as the first argument.

Parameters

- **items** — *array / element / Muuri.Item / number / string*
  - To target specific items provide an array of item instances/elements/indices. By default all active items are targeted.
  - Optional.

```
// Refresh dimensions of all active item elements.
grid.refreshItems();

// Refresh dimensions of specific item elements.
grid.refreshItems([0, someElem, someItem]);
```

grid.refreshSortData([items])

Refresh the sort data of the instance's items.

Parameters

- **items** — *array / element / Muuri.Item / number*
  - To target specific items provide an array of item instances/elements/indices. By default all items are targeted.
  - Optional.

```
// Refresh the sort data for every item.
grid.refreshSortData();

// Refresh the sort data for specific items.
grid.refreshSortData([0, someElem, someItem]);
```

grid.synchronize()

Synchronize the item elements to match the order of the items in the DOM. This comes handy if you need to keep the DOM structure matched with the order of the items. Note that if an item's element is not currently a child of the container element (if it is dragged for example) it is ignored and left untouched.

```
// Let's say we have to move the first item in the grid as the last.
grid.move(0, -1);
// Now the DOM order of the items is not in sync anymore with the
// order of the items. We can sync the DOM with synchronize method.
grid.synchronize();
```

grid.layout([instant], [callback])

Calculate item positions and move items to their calculated positions, unless they are already positioned correctly. The grid's height/width (depends on the layout algorithm) is also adjusted according to the position of the items.

Parameters

- **instant** — *boolean*
  - Should the items be positioned instantly without any possible animation?
  - Default value: `false`.
  - Optional.
- **callback** — *function*
  - A callback function that is called after the items have positioned. Receives one argument: an array of all the items that were successfully positioned without interruptions.
  - Optional.

```
// Layout items.
grid.layout();

// Layout items instantly (without animations).
grid.layout(true);

// Layout all items and define a callback that will be called
// after all items have been animated to their positions.
grid.layout(function (items) {
  console.log('layout done!');
});
```

grid.add(elements, [options])

Add new items by providing the elements you wish to add to the instance and optionally provide the index where you want the items to be inserted into. All elements that are not already children of the container element will be automatically appended to the container element. If an element has it's CSS display property set to none it will be marked as *inactive* during the initiation process. As long as the item is *inactive* it will not be part of the layout, but it will retain it's index. You can activate items at any point with `grid.show()` method. This method will automatically call `grid.layout()` if one or more of the added elements are visible. If only hidden items are added no layout will be called. All the new visible items are positioned without animation during their first layout.

Parameters

- **elements** — *array / element*
  - An array of DOM elements.
- **options.index** — *number*
  - The index where you want the items to be inserted in. A value of `-1` will insert the items to the end of the list while `0` will insert the items to the beginning. Note that the DOM elements are always just appended to the instance container regardless of the index value. You can use the `grid.synchronize()` method to arrange the DOM elments to the same order as the items.
  - Default value: `-1`.
  - Optional.
- **options.layout** — *boolean / function / string*
  - By default `grid.layout()` is called at the end of this method. With this argument you can control the layout call. You can disable the layout completely with `false`, or provide a callback function for the layout method, or provide the string `'instant'` to make the layout happen instantly without any animations.
  - Default value: `true`.
  - Optional.

Returns — *array*

Returns the added items.

```
// Add two new items to the end.
grid.add([elemA, elemB]);

// Add two new items to the beginning.
grid.add([elemA, elemB], {index: 0});

// Skip the automatic layout.
grid.add([elemA, elemB], {layout: false});
```

grid.remove(items, [options])

Remove items from the instance.

Parameters

- **items** — *array / element / Muuri.Item / number*
  - An array of item instances/elements/indices.
- **options.removeElements** — *boolean*
  - Should the associated DOM element be removed from the DOM?
  - Default value: `false`.

  - Optional.
- **options.layout** — *boolean / function / string*
  - By default `grid.layout()` is called at the end of this method. With this argument you can control the layout call. You can disable the layout completely with `false`, or provide a callback function for the layout method, or provide the string `'instant'` to make the layout happen instantly without any animations.
  - Default value: `true`.
  - Optional.

Returns — *array*

Returns the destroyed items.

```
// Remove the first item, but keep the element in the DOM.
grid.remove(0);

// Remove items and the associated elements.
grid.remove([elemA, elemB], {removeElements: true});

// Skip the layout.
grid.remove([elemA, elemB], {layout: false});
```

grid.show(items, [options])

Show the targeted items.

Parameters

- **items** — *array / element / Muuri.Item / number*
  - An array of item instances/elements/indices.
- **options.instant** — *boolean*
  - Should the items be shown instantly without any possible animation?
  - Default value: `false`.
  - Optional.
- **options.onFinish** — *function*
  - A callback function that is called after the items are shown.
  - Optional.
- **options.layout** — *boolean / function / string*
  - By default `grid.layout()` is called at the end of this method. With this argument you can control the layout call. You can disable the layout completely with `false`, or provide a callback function for the layout method, or provide the string `'instant'` to make the layout happen instantly without any animations.
  - Default value: `true`.
  - Optional.

```
// Show items with animation (if any).
grid.show([elemA, elemB]);

// Show items instantly without animations.
grid.show([elemA, elemB], {instant: true});

// Show items with callback (and with animations if any).
grid.show([elemA, elemB], {onFinish: function (items) {
  console.log('items shown!');
}});
```

grid.hide(items, [options])

Hide the targeted items.

Parameters

- **items** — *array / element / Muuri.Item / number*
  - An array of item instances/elements/indices.
- **options.instant** — *boolean*
  - Should the items be hidden instantly without any possible animation?

    - Default value: `false` .
    - Optional.
- **options.onFinish** — *function*
    - A callback function that is called after the items are hidden.
    - Optional.
- **options.layout** — *boolean / function / string*
    - By default `grid.layout()` is called at the end of this method. With this argument you can control the layout call. You can disable the layout completely with `false` , or provide a callback function for the layout method, or provide the string `'instant'` to make the layout happen instantly without any animations.
    - Default value: `true` .
    - Optional.

```
// Hide items with animation.
grid.hide([elemA, elemB]);

// Hide items instantly without animations.
grid.hide([elemA, elemB], {instant: true});

// Hide items and call the callback function after
// all items are hidden.
grid.hide([elemA, elemB], {onFinish: function (items) {
  console.log('items hidden!');
}});
```

grid.filter(predicate, [options])

Filter items. Expects at least one argument, a predicate, which should be either a function or a string. The predicate callback is executed for every item in the instance. If the return value of the predicate is truthy the item in question will be shown and otherwise hidden. The predicate callback receives the item instance as it's argument. If the predicate is a string it is considered to be a selector and it is checked against every item element in the instance with the native element.matches() method. All the matching items will be shown and others hidden.

Parameters

- **predicate** — *function / string*
    - A predicate callback or a selector.
- **options.instant** — *boolean*
    - Should the items be shown/hidden instantly without any possible animation?
    - Default value: `false` .
    - Optional.
- **options.onFinish** — *function*
    - An optional callback function that is called after all the items are shown/hidden.
    - Optional.
- **options.layout** — *boolean / function / string*
    - By default `grid.layout()` is called at the end of this method. With this argument you can control the layout call. You can disable the layout completely with `false` , or provide a callback function for the layout method, or provide the string `'instant'` to make the layout happen instantly without any animations.
    - Default value: `true` .
    - Optional.

```
// Show all items that have the attribute "data-foo".
grid.filter(function (item) {
  return item.getElement().hasAttribute('data-foo');
});

// Or simply just...
grid.filter('[data-foo]');

// Show all items that have a class foo.
grid.filter('.foo');
```

grid.sort(comparer, [options])

Sort items. There are three ways to sort the items. The first is simply by providing a function as the comparer which works almost identically to native array sort. Th only difference is that the sort is always stable. Alternatively you can sort by the sort data you have provided in the instance's options. Just provide the sort data key(s) as a string (separated by space) and the items will be sorted based on the provided sort data keys. Lastly you have the opportunity to provide a presorted array of items which will be used to sync the internal items array in the same order.

Parameters

- **comparer** — *array / function / string*
  - Provide a comparer function, sort data keys as a string (separated with space) or a presorted array of items. It is recommended to use the sort data feature, because it allows you to cache the sort data and make the sorting faster.
- **options.descending** — *boolean*
  - By default the items are sorted in ascending order. If you want to sort them in descending order set this to `true` .
  - Default value: `false` .
  - Optional.
- **options.layout** — *boolean / function / string*
  - By default `grid.layout()` is called at the end of this method. With this argument you can control the layout call. You can disable the layout completely with `false` , or provide a callback function for the layout method, or provide the string `'instant'` to make the layout happen instantly without any animations.
  - Default value: `true` .
  - Optional.

```
// Sort items by data-id attribute value (ascending).
grid.sort(function (itemA, itemB) {
  var aId = parseInt(itemA.getElement().getAttribute('data-id'));
  var bId = parseInt(itemB.getElement().getAttribute('data-id'));
  return aId - bId;
});

// Sort items with a presorted array of items.
grid.sort(presortedItems);

// Sort items using the sort data keys (ascending).
grid.sort('foo bar');

// Sort items using the sort data keys (descending).
grid.sort('foo bar', {descending: true});

// Sort items using the sort data keys. Sort some keys
// ascending and some keys descending.
grid.sort('foo bar:desc');
```

grid.move(item, position, [options])

Move an item to another position in the grid.

Parameters

- **item** — *element / Muuri.Item / number*
  - Item instance, element or index.
- **position** — *element / Muuri.Item / number*
  - Item instance, element or index.
- **options.action** — *string*
  - Accepts the following values:
    - `'move'` : moves item in place of another item.
    - `'swap'` : swaps position of items.
  - Default value: `'move'` .
  - Optional.
- **options.layout** — *boolean / function / string*
  - By default `grid.layout()` is called at the end of this method. With this argument you can control the layout call. You can disable the layout completely with `false` , or provide a callback function for the layout method, or provide the string `'instant'` to make the layout happen instantly without any animations.
  - Default value: `true` .
  - Optional.

```
// Move elemA to the index of elemB.
grid.move(elemA, elemB);

// Move the first item in the grid as the last.
grid.move(0, -1);

// Swap positions of elemA and elemB.
grid.move(elemA, elemB, {action: 'swap'});

// Swap positions of the first and the last item.
grid.move(0, -1, {action: 'swap'});
```

grid.send(item, grid, position, [options])

Move an item into another grid.

Parameters

- **item** — *element / Muuri.Item / number*
  - The item that should be moved. You can define the item with an item instance, element or index.
- **grid** — *Muuri*
  - The grid where the item should be moved to.
- **position** — *element / Muuri.Item / number*
  - To which position should the item be placed to in the new grid? You can define the position with an item instance, element or index.
- **options.appendTo** — *element*
  - Which element the item element should be appended to for the duration of the layout animation?
  - Default value: `document.body` .
- **options.layoutSender** — *boolean / function / string*
  - By default `grid.layout()` is called for the sending grid at the end of this method. With this argument you can control the layout call. You can disable the layout completely with `false` , or provide a callback function for the layout method, or provide the string `'instant'` to make the layout happen instantly without any animations.
  - Default value: `true` .
  - Optional.
- **options.layoutReceiver** — *boolean / function / string*
  - By default `grid.layout()` is called for the receiving grid at the end of this method. With this argument you can control the layout call. You can disable the layout completely with `false` , or provide a callback function for the layout method, or provide the string `'instant'` to make the layout happen instantly without any animations.
  - Default value: `true` .
  - Optional.

```
// Move the first item of gridA as the last item of gridB.
gridA.send(0, gridB, -1);

// Move the first item of gridA as the last item of gridB.
gridA.send(0, gridB, -1 {
  appendTo: someElem
});

// Do something after the item has been sent and the layout
// processes have finished.
gridA.send(0, gridB, -1 {
  layoutSender: function (isAborted, items) {
    // Do your thing here...
  },
  layoutReceiver: function (isAborted, items) {
    // Do your other thing here...
  }
});
```

grid.on(event, listener)

Bind an event listener.

Parameters

- **event** — *string*
- **listener** — *function*

Returns — *object*

Returns the instance.

```
grid.on('layoutEnd', function (items) {
  console.log(items);
});
```

grid.once(event, listener)

Bind an event listener that is triggered only once.

Parameters

- **event** — *string*
- **listener** — *function*

Returns — *object*

Returns the instance.

```
grid.once('layoutEnd', function (items) {
  console.log(items);
});
```

grid.off(event, listener)

Unbind an event listener.

Parameters

- **event** — *string*
- **listener** — *function*

Returns — *object*

Returns the instance.

```
var listener = function (items) {
  console.log(items);
};

muuri
.on('layoutEnd', listener)
.off('layoutEnd', listener);
```

grid.destroy([removeElements])

Destroy the grid instance.

Parameters

- **removeElements** — *boolean*
  - Should the item elements be removed or not?
  - Default value: `false` .
  - Optional.

Returns — *object*

Returns the instance.

```
// Destroy the instance.
grid.destroy();
```

```
// Destroy the instance and remove item elements.
grid.destroy(true);
```

Grid events

- synchronize
- layoutStart
- layoutEnd
- add
- remove
- showStart
- showEnd
- hideStart
- hideEnd
- filter
- sort
- move
- send
- beforeSend
- receive
- beforeReceive
- dragInit
- dragStart
- dragMove
- dragScroll
- dragEnd
- dragReleaseStart
- dragReleaseEnd
- destroy

synchronize

Triggered after `grid.synchronize()` is called.

```
grid.on('synchronize', function () {
  console.log('Synced!');
});
```

layoutStart

Triggered after `grid.layout()` is called, just before the items are positioned.

Arguments

- **items** — *array*
  - The items that are about to be positioned.

```
grid.on('layoutStart', function (items) {
  console.log(items);
});
```

layoutEnd

Triggered after `grid.layout()` is called, after the items have positioned. Note that if `grid.layout()` is called during an ongoing layout animation the ongoing layout process will be aborted and it's layoutEnd event will never be triggered.

Arguments

- **items** — *array*

  - The items that were intended to be positioned. Note that these items are always identical to what the layoutStart event's callback receives as it's argument. So if, for example, you destroy an item during the layout animation and don't do call another layout the destroyed item will still be included in this array of items. The original intention was to filter these items so that all items that were "interrupted" somehow during the layout process would be omitted from the results, but that solution was much more prone to errors and much more harder to explain/understand.

```
grid.on('layoutEnd', function (items) {
  console.log(items);
});
```

add

Triggered after `grid.add()` is called.

Arguments

- **items** — *array*
  - The items that were succesfully added.

```
grid.on('add', function (items) {
  console.log(items);
});
```

remove

Triggered after `grid.remove()` is called.

Arguments

- **indices** — *array*
  - Indices of the items that were succesfully removed.

```
grid.on('remove', function (indices) {
  console.log(indices);
});
```

showStart

Triggered after `grid.show()` is called, just before the items are shown.

Arguments

- **items** — *array*
  - The items that are about to be shown.

```
grid.on('showStart', function (items) {
  console.log(items);
});
```

showEnd

Triggered after `grid.show()` is called, after the items are shown.

Arguments

- **items** — *array*
  - The items that were succesfully shown without interruptions. If you, for example, call `grid.hide()` to some of the items that are currently being shown, those items will be omitted from this argument.

```
grid.on('showEnd', function (items) {
  console.log(items);
});
```

hideStart

Triggered after `grid.hide()` is called, just before the items are hidden.

Arguments

- **items** — *array*
    - The items that are about to be hidden.

```
grid.on('hideStart', function (items) {
  console.log(items);
});
```

hideEnd

Triggered after `grid.hide()` is called, after the items are hidden.

Arguments

- **items** — *array*
    - The items that were succesfully hidden without interruptions. If you, for example, call `grid.show()` to some of the items that are currently being hidden, those items will be omitted from this argument.

```
grid.on('hideEnd', function (items) {
  console.log(items);
});
```

filter

Triggered after `grid.filter()` is called.

Arguments

- **shownItems** — *array*
    - The items that are shown.
- **hiddenItems** — *array*
    - The items that are hidden.

```
grid.on('filter', function (shownItems, hiddenItems) {
  console.log(shownItems);
  console.log(hiddenItems);
});
```

sort

Triggered after `grid.sort()` is called.

Arguments

- **currentOrder** — *array*
    - All items in their current order.
- **previousOrder** — *array*
    - All items in their previous order.

```
grid.on('sort', function (currentOrder, previousOrder) {
  console.log(currentOrder);
  console.log(previousOrder);
});
```

move

Triggered after `grid.move()` is called or when the grid is sorted during drag. Note that this is event not triggered when an item is dragged into another grid.

Arguments

- **data** — *object*

- **data.item** — *Muuri.Item*
  - The item that was moved.
- **data.fromIndex** — *number*
  - The index the item was moved from.
- **data.toIndex** — *number*
  - The index the item was moved to.
- **data.action** — *string*
  - "move" or "swap".

```
grid.on('move', function (data) {
  console.log(data);
});
```

send

Triggered for the originating grid in the end of the *send process* (after `grid.send()` is called or when an item is dragged into another grid). Note that this event is called *before* the item's layout starts.

Arguments

- **data** — *object*
  - **data.item** — *Muuri.Item*
    - The item that was sent.
  - **data.fromGrid** — *Muuri*
    - The grid the item was sent from.
  - **data.fromIndex** — *number*
    - The index the item was moved from.
  - **data.toGrid** — *Muuri*
    - The grid the item was sent to.
  - **data.toIndex** — *number*
    - The index the item was moved to.

```
grid.on('send', function (data) {
  console.log(data);
});
```

beforeSend

Triggered for the originating grid in the beginning of the *send process* (after `grid.send()` is called or when an item is dragged into another grid). This event is highly useful in situations where you need to manipulate the sent item (freeze it's dimensions for example) before it is appended to it's temporary layout container as defined in send method options.

Arguments

- **data** — *object*
  - **data.item** — *Muuri.Item*
    - The item that was sent.
  - **data.fromGrid** — *Muuri*
    - The grid the item was sent from.
  - **data.fromIndex** — *number*
    - The index the item was moved from.
  - **data.toGrid** — *Muuri*
    - The grid the item was sent to.
  - **data.toIndex** — *number*
    - The index the item was moved to.

```
grid.on('beforeSend', function (data) {
  console.log(data);
});
```

receive

Triggered for the receiving grid in the end of the *send process* (after `grid.send()` is called or when an item is dragged into another grid). Note that this event is called *before* the item's layout starts.

Arguments

- **data** — *object*
  - **data.item** — *Muuri.Item*
    - The item that was sent.
  - **data.fromGrid** — *Muuri*
    - The grid the item was sent from.
  - **data.fromIndex** — *number*
    - The index the item was moved from.
  - **data.toGrid** — *Muuri*
    - The grid the item was sent to.
  - **data.toIndex** — *number*
    - The index the item was moved to.

```
grid.on('receive', function (data) {
  console.log(data);
});
```

beforeReceive

Triggered for the receiving grid in the beginning of the *send process* (after `grid.send()` is called or when an item is dragged into another grid). This event is highly useful in situations where you need to manipulate the received item (freeze it's dimensions for example) before it is appended to it's temporary layout container as defined in send method options.

Arguments

- **data** — *object*
  - **data.item** — *Muuri.Item*
    - The item that was sent.
  - **data.fromGrid** — *Muuri*
    - The grid the item was sent from.
  - **data.fromIndex** — *number*
    - The index the item was moved from.
  - **data.toGrid** — *Muuri*
    - The grid the item was sent to.
  - **data.toIndex** — *number*
    - The index the item was moved to.

```
grid.on('beforeReceive', function (data) {
  console.log(data);
});
```

dragInit

Triggered in the beginning of the *drag start* process when dragging of an item begins. This event is highly useful in situations where you need to manipulate the dragged item (freeze it's dimensions for example) before it is appended to the dragContainer.

Arguments

- **item** — *Muuri.Item*
  - The dragged item.
- **event** — *object*
  - Hammer.js event data.

```
grid.on('dragInit', function (item, event) {
  console.log(event);
  console.log(item);
});
```

dragStart

Triggered in the end of the *drag start* process when dragging of an item begins.

Arguments

- **item** — *Muuri.Item*
  - The dragged item.
- **event** — *object*
  - Hammer.js event data.

```
grid.on('dragStart', function (item, event) {
  console.log(event);
  console.log(item);
});
```

dragMove

Triggered when an item is dragged.

Arguments

- **item** — *Muuri.Item*
  - The dragged item.
- **event** — *object*
  - Hammer.js event data.

```
grid.on('dragMove', function (item, event) {
  console.log(event);
  console.log(item);
});
```

dragScroll

Triggered when any of the scroll parents of a dragged item is scrolled.

Arguments

- **item** — *Muuri.Item*
  - The dragged item.
- **event** — *object*
  - The scroll event data.

```
grid.on('dragScroll', function (item, event) {
  console.log(event);
  console.log(item);
});
```

dragEnd

Triggered when dragging of an item ends.

Arguments

- **item** — *Muuri.Item*
  - The dragged item.
- **event** — *object*
  - Hammer.js event data.

```
grid.on('dragEnd', function (item, event) {
  console.log(event);
  console.log(item);
});
```

dragReleaseStart

Triggered when a dragged item is released.

Arguments

- **item** — *Muuri.Item*
    - The released item.

```
grid.on('dragReleaseStart', function (item) {
  console.log(item);
});
```

dragReleaseEnd

Triggered after released item has been animated to position.

Arguments

- **item** — *Muuri.Item*
    - The released item.

```
grid.on('dragReleaseEnd', function (item) {
  console.log(item);
});
```

destroy

Triggered after `grid.destroy()` is called.

```
grid.on('destroy', function () {
  console.log('Muuri is no more...');
});
```

Item methods

- item.getGrid()
- item.getElement()
- item.getWidth()
- item.getHeight()
- item.getMargin()
- item.getPosition()
- item.isActive()
- item.isVisible()
- item.isShowing()
- item.isHiding()
- item.isPositioning()
- item.isDragging()
- item.isReleasing()
- item.isDestroyed()

item.getGrid()

Get the instance's grid instance.

Returns — *Muuri*

```
var grid = item.getMuuri();
```

item.getElement()

Get the instance element.

Returns — *element*

```
var elem = item.getElement();
```

item.getWidth()

Get instance element's cached width. The returned value includes the element's paddings and borders. Note that the values are rounded with `Math.round()` .

Returns — *number*

```
var width = item.getWidth();
```

item.getHeight()

Get instance element's cached height. The returned value includes the element's paddings and borders. Note that the values are rounded with `Math.round()` .

Returns — *number*

```
var height = item.getHeight();
```

item.getMargin()

Get instance element's cached margins. Note that the values are rounded with `Math.round()` .

Returns — *object*

- **obj.left** — *number*
- **obj.right** — *number*
- **obj.top** — *number*
- **obj.bottom** — *number*

```
var margin = item.getMargin();
```

item.getPosition()

Get instance element's cached position (relative to the container element).

Returns — *object*

- **obj.left** — *number*
- **obj.top** — *number*

```
var position = item.getPosition();
```

item.isActive()

Check if the item is currently *active*. Only active items are considered to be part of the layout.

Returns — *boolean*

```
var isActive = item.isActive();
```

item.isVisible()

Check if the item is currently *visible*.

Returns — *boolean*

```
var isVisible = item.isVisible();
```

item.isShowing()

Check if the item is currently animating to visible.

Returns — *boolean*

```
var isShowing = item.isShowing();
```

item.isHiding()

Check if the item is currently animating to hidden.

Returns — *boolean*

```
var isHiding = item.isHiding();
```

item.isPositioning()

Check if the item is currently being positioned.

Returns — *boolean*

```
var isPositioning = item.isPositioning();
```

item.isDragging()

Check if the item is currently being dragged.

Returns — *boolean*

```
var isDragging = item.isDragging();
```

item.isReleasing()

Check if the item is currently being released.

Returns — *boolean*

```
var isReleasing = item.isReleasing();
```

item.isDestroyed()

Check if the item is destroyed.

Returns — *boolean*

```
var isDestroyed = item.isDestroyed();
```

FAQ

Can you help me with ...?

First of all you should check out the current questions and see if your question has been asked/answered already. If not, you can create create a new issue and explain your problem.

I think I found a bug, what should I do?

Please create an issue and explain the bug in detail. If possible create a reduced test case and share a link to it. You can, for example, fork this CodePen example and modify it to demonstrate the bug.

Is there a React/Vue version?

Not yet, but it is planned. Hold on tight!

Credits

Created and maintained by Niklas Rämö.

- This project owes much to David DeSandro's Masonry and Packery libraries. You should go ahead and check them out right now if you haven't yet. Thanks Dave!
- Jukka Jylänki's research A Thousand Ways to Pack the Bin came in handy when building Muuri's layout algorithms. Thanks Jukka!
- Big thanks to the people behind Velocity.js and Hammer.js for providing such awesome libraries. Muuri would be much less cool without animations and dragging.
- Haltu Oy was responsible for initiating this project in the first place and funded the intial development. Thanks Haltu!

License