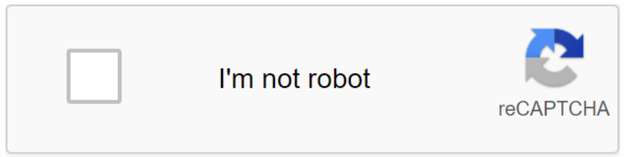

Continue

A complete guide to flexbox

The Flexbox Layout (Flexible Box) module (a W3C Candidate recommendation as of October 2017) aims to provide a more efficient way to arrange, align, and distribute space between elements in a container, even when their size is unknown and/or dynamic (hence the word flex). The main idea behind flex layout is to give the container the ability to change the width/height of its elements (and order) to best fill the available space (mostly to accommodate all kinds of display devices and screen sizes). A flex container expands items to fill available free space or reduces them to avoid overflow. More importantly, the flexbox layout is direction-independent compared to normal layouts (block that is based vertically and inline which is based horizontally). While these work well for pages, they don't have flexibility (no intended pun) to support large or complex applications (especially when it comes to orientation change, scaling, stretching, shrinking, etc.). Note: Flexbox layout is more appropriate for application components and small-scale layouts, while grid layout is intended for larger-scale layouts. Since flexbox is an entire module and not a single property, it involves a lot of things including its entire set of properties. Some of them are intended to be set on the container (parent element, known as flexible container) while the others are intended to be set on child elements (called flex elements). If the normal layout is based on both lock and flow inline directions, the flex layout is based on flexible flow directions. Please have a look at this figure from the specification, explaining the main idea behind the flex layout. The elements will be arranged following the main axis (from the main beginning to the main end) or the transverse axis (from the cross-start to the cross-end). Main axis – The main axis of a flex container is the primary axis along which flex elements are arranged. Attention, it is not necessarily horizontal; depends on the flex-direction property (see below).main-start main-end – Flex elements are placed inside the container starting at the main beginning and going to main-end.main size – The width or height of a flex element, depending on the main size, is the main size of the item. The main size property of the flex element is the 'width' or 'height' property, depending on the main size. Its direction depends on the direction of the main axis. cross-end – Flex lines are filled with the elements and inserted into the starting from the cross-start side of the flex container and towards the cross-end.cross size side – The width or height of a flex element, depending on the transverse size, is the transverse dimension of the element. The cross size property is any of 'width' or 'height' that is in the transverse dimension. Cross. this guide a lot? Add a copy to the office wall. This defines a flex container; inline or block depending on the specified value. It allows a flex context for all its direct children. .container : display: flex; / or inline-flex - / - Note that CSS columns have no effect on a flex container. This establishes the main axis, thus defining the direction in which flex elements are placed in the flex container. Flexbox is (apart from optional wrapping) a single-direction layout concept. Think of flex elements as primarily layout in horizontal rows or vertical columns. .container : flex-direction: row - row-reverse , column , column-inversion; line (default): left to right in ltr; right to left in rtlrow-reverse: right to left in ltr; left to right in rtlcolumn: equal to row but top to bottomcolumn-reverse: equal to line-reverse but bottom-to-top By default, flex elements will try to fit a row. You can change this setting and allow items to return to carriage as needed with this property. .container : flex-wrap: nowrap , wrap, wrap-reverse; , nowrap (default): All flex elements will be on a single line: flex elements will be wrapped on multiple lines, top to bottom.wrap-reverse: flex elements will be wrapped on multiple lines from bottom to top. There are some visual flex-wrap demos here. It is an abbreviation for the flex-direction and flex-wrap properties, which together define the main and transverse axes of the flex container. The default value is row nowrap. .container : flexible flow: carriage return of the column; Defines the alignment along the main axis. It helps to distribute additional free space when all flex elements on a line are inflexible or flexible but have reached their maximum size. It also exerts some control over the alignment of elements when they cross the line. .container - justify-content: flex-start, flex-end, center, space-around, space-around, space-uniform, beginning, end, left, right ... , safe, unsafe; - flex-start (default): the elements are compressed towards the beginning of flex-direction.flex-end: the elements are compressed towards the end of flex-direction.start: the elements are compressed towards the beginning of the direction of the writing mode.end: the elements are compressed towards the end of the direction of the writing mode.left: the elements are compressed towards the left edge of the container, unless this makes sense with flex direction, then it behaves like start.right: the elements packed towards the right edge of the container, unless this makes sense with the flex direction, then it behaves like start.center: the elements are centered along the line space between: the elements are evenly distributed in the row; the first item is on the start line, the last item on the final line: items are evenly distributed on the line with the same space around them. Note that visually the spaces are not the same, since all elements have the same space on both sides. The first element will have a unit of space space border, but two units of space between the next element because the next element has its own spacing that applies.space-evenly: the elements are distributed so that the spacing between any two elements (and the space for the edges) is the same. Note that browser support for these values is nuanced. For example, the space between has never gotten support for some versions of Edge, and start/end/left/right aren't in Chrome yet. MDN has detailed charts. The safest values are flex-start, flex-end and center. There are also two additional keywords that you can associate with these values: safe and unsafe. Using safe ensures that, however you perform this type of placement, you cannot push an element so that it renders off-screen (e.g. off-screen) so that even the content cannot be scrolled (called data loss). Defines the default behavior for how flex elements are arranged along the transverse axis on the current line. Think of it as the justify-content version for the transverse axis (perpendicular to the main axis). .container - align-items: stretch, flex-start, flex-end, center, baseline, last baseline, beginning, beginning and end, auto-end, ... Safe. unsafe; - stretch (default): stretch to fill the container (still respect min-width/max-width)flex-start / start / self-start: the elements are placed at the beginning of the transverse axis. The difference between these is subtle, and it's about complying with flex direction rules or write mode rules.flex-end/end/self-end: elements are placed at the end of the transverse axis. The difference is once again subtle and concerns compliance with flexible direction rules with respect to writing mode rules.center: elements are centered in the base between axes: elements are aligned as their baselines align Safe and unsafe modifier keywords can be used in combination with all other keywords (although browser support known) and allow you to avoid aligning elements so that content becomes inaccessible. This way the lines of a flex container within when there is additional space in the transverse axis, similar to how justify-content aligns individual elements within the main axis. Note: This property affects only flexible containers on multiple lines, where flex-flow is set to wrap or wrap-reverse). A flexible single-line container (i.e. when flex-flow is set to the default value, no-wrap) will not reflect align-content. .container - align-content: flex-start , flex-end , center , , space-around, space-uniform, stretch, start, end, first baseline, last baseline ... Safe. unsafe; - normal (default): Items are compressed to their default location as if no value had been set. The flex-start (more supported) respects the flex direction while the start respects the direction of the writing mode. The (plus (plus) flex-end respects flex direction while end respects the direction of write mode.center: elements centered in container-between space: evenly distributed elements; the first row is at the beginning of the container, while the last line is at the end of the space: the elements evenly distributed with the same space around each linespace-uniformly: the elements are evenly distributed with the same space around themstretch: the lines extend To take up the remaining space Safe and unsafe editing keywords can be used in combination with all other keywords (although browser support notes) and take care of helping prevent item alignment so that content becomes inaccessible. By default, flex elements are arranged in the source order. However, the order property controls the order in which they are displayed in the flex container. .item : order: 5; / / default is 0 / - This defines the possibility for a flex element to grow if necessary. Accepts a unitless value that acts as a proportion. Determines the amount of available space within the flex container that the element should take. If all elements have flex-grow set to 1, the remaining space in the container will be distributed equally to all child elements. If one of the children has a value of 2, the remaining space would require twice the space of the others (or try to do so, at least). .item : flex-grow: 4; default 0 s/: Negative numbers are invalid. This defines the possibility for a flex element to shrink if necessary. .item : flex-shrink: 3; default 1 s/: Negative numbers are invalid. Defines the default size of an item before the remaining space is distributed. It can be a length (e.g. 20%, 5rem, etc.) or a keyword. The auto keyword means looking at my width or height property (which was temporarily run from the main-size keyword until it is deprecated). The content keyword means size it based on the item's content – this keyword is not yet well supported, so it's hard to test and harder to know what its max-content, min-content, and fit-content siblings do... the element: flex-basis: car; / / default car, if set to 0, the additional space around the content is not taken into account. When set to a car, the additional space is distributed based on its flexibility value. See this chart. This is short for flex-grow, flex-shrink and flex-basis combined. The second and third parameters (flex-shrink and flex-basis) are optional. The default value is 0 1 car, but if you set it with a single value it's like 1 0. .item : flex: none [< 'flex-grow'> <'flex-shrink'>? <'flex-basis'>] We recommend that you use this abbreviated property instead of setting individual properties. The abbreviation sets the other values intelligently. This allows you to perform the default alignment (or the one specified by align-items) for individual flex items. Please refer to the align-items explanation to understand understand .item : align-self: car - flex-start - flex-end - center - baseline - stretch; Note that float, clear, and vertical-align have no effect on a flex element. Let's start with a very simple example, solving an almost daily problem: perfect centering. It couldn't be easier if you use flexbox. .parent : display: flex; height: 300px; / or whatever whatever .child :width: 100px; / or whatever the height: 100px; / or whatever the margin: car; Magic! Then setting a vertical car margin will make the element perfectly centered in both axes. Now let's use other properties. Consider a list of 6 items, all with fixed sizes, but they can be resized automatically. We want them to be evenly distributed on the horizontal axis so that when we resize the browser, everything scales well and without media queries. .flex-container - / let's first create a flex layout context : / display: flex; / Then we define the direction of the flow and if we allow the elements to wrap: Remember that this is the same as: - flex-direction: row; Done. Everything else is just some kind of style concern. Below is a pen with this example. Be sure to go to CodePen and try to resize the windows to see what happens. Let's try something else. Imagine you have a right-aligned navigation element at the top of our website, but we want it to be centered on medium-sized, single-column screens on small devices. Pretty easy. .navigation : display: flex; flex-flow: return to line head; / / This aligns the elements to the final line on the main axis - / justify-content: flex-end; / / medium screens / @media all e (max-width: 800px). in the center, distributing @media evenly distribute the empty space around the elements: // justify-content: space-around; Let's try something even better by playing with the flexibility of flex elements! What about a mobile-first 3-column layout with full-width header and footer. It is independent of the order of origin. .wrapper : display: flex; flex-flow: row wrap; / / / . > In this case, you rely on the source order for the mobile-first approach. heading No 2. article 3. apart from 1 and 4. apart from 2 - 5. footer / / / medium screens @media all and (min-width: 600px) / / / We tell both sidebars to share a row @media. two side bars : .main : flex: 2 0px; .aside-1 - order: 1; .main . Flexbox requires some vendor prefixes to support as many browsers as possible. It not only includes properties of prepending the vendor prefix, but actually there are completely different property names and values. This is because the Flexbox specification has changed over time, creating an old, tweener and new version. Perhaps the best way to handle this is to write in the new (and final) syntax and run the CSS via Autoprefixer, which handles fallbacks very well. Alternatively, here is a sass @mixin to help with some of the prefixes, which also gives you an idea of what kind of things need to be done: @mixin flexbox() : display: -webkit-box; display: -moz-box; display: -ms-flexbox; display: -webkit-flex; display $values: flex $values $values @mixin; -webkit-flex: $values; -ms-flex: $values; flex: $values; - @mixin order ($val) - -webkit-box-ordinal-group: $val; -moz-box-ordinal-group: $val; -ms-flex-order: $val; -webkit-order: $val; order: $val; .wrapper @include flexbox(@include @include); Flexbox is certainly not without its bugs. The best collection I've ever seen is Flexbugs by Philip Walton and Greg Whitworth. It's an open source place to track them all down, so I think it's best to link to that. Broken down by flexbox version: (new) means that the recent syntax from the specification (e.g. display: flex;)(tweener) means a strange unofficial syntax since 2011 (e.g. display: flexbox;)(old) means the old syntax of 2009 (e.g. display: box;) ChromeSafariFirefoxOperaIEEdgeAndroidiOS20- (old)21 ' (new)3.1 (old)6.1 (new)2-21 (old)22' (new)12.1 (new))1 10 (tweener)11 (new)17 (new)2.1 (old)4.4 (new)3.2 (old)7.1 (new) Blackberry browser 10 supports the new syntax. Syntax.