**CSS Layout Managers: float, flexbox, grid**

So far we have seen how to create different HTML elements and style them using CSS. However, in many modern HTML pages, you also need to place the HTML elements at specific positions on the screen (layout), and also change the page layout according to different screen sizes called the ***responsive design***. For example, in a big screen you may want, say, 4 HTML elements, e.g., images, to appear side-by-side on the screen, but on a mobile phone with a small screen you may want the images to appear vertically one after the other. You want to achieve all this with CSS without changing the content of your HTML page. This is the essence of *responsive design*.

Traditionally, HTML elements are placed either on the same row (inline elements), or vertically one after the other (block-level elements). With new CSS layout managers, it is possible to create arbitrary, multi-column layouts.

There are five different ways to create *multicolumn* layouts. Each way has its pros and cons:

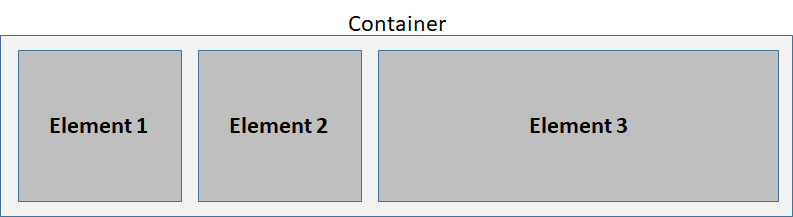
* **HTML tables**: Old way of laying out HTML elements. This is not recommended anymore.
* **CSS float property**: It is common to do entire web layouts using the CSS float property. Float is easy to learn - you just need to remember how the *float* and *clear* properties work. Disadvantages: Floating elements are tied to the document flow, which may harm the flexibility.
* **CSS flexbox**: Flexbox is a new layout mode in CSS3. Use of flexbox ensures that elements behave predictably when the page layout must accommodate different screen sizes and different display devices. Disadvantages: Does not work in IE10 and earlier.
* **CSS grid**: The CSS Grid Layout Module offers a grid-based layout system, with rows and columns, making it easier to design web pages without having to use floats and positioning. Disadvantages: Does not work in IE nor in Edge 15 and earlier.
* **CSS frameworks**: Bootstrap etc. Makes responsive layout designs much easier. Very popular.

**CSS float/clear Property**

The **float** property was originally designed to be used to wrap text around images:

Look at: 03-Layout/01-Float1

Using the **float** property, it is possible to stack elements side-by-side in a multi-column. Assume that you have several elements that needs to be stacked side-by-side in a multi-column format in a container, then you can use the float property as follows:



In this example, we have 3 elements. Element1 and Element2 each occupies 25% of the container’s total width, and Element3 occupies 50%.

The float property can have one of the following values:

* left - The element floats to the left of its container
* right - The element floats to the right of its container
* none - The element does not float (will be displayed just where it occurs in the text). This is the default
* inherit - The element inherits the float value of its parent

Look at: 03-Layout/02-Float2

Notice from the example that it is possible to make the width of the elements inside a container the same, but it is not possible to make their heights the same. If we fix the height to some fixed value in pixels, then the content that does not fit inside the container will overflow! We can fix this problem using the flexbox model, which we will cover next.

The **clear** property is used to clear the current float style. In other words, you can specify that the elements that come after clear should not float to the left or right but should go to the next line. Unless you clear the existing float, the coming elements will continue to float on the current line.

The **clear** property can have one of the following values:

* none - Allows floating elements on both sides. This is the default
* left - No floating elements allowed on the left side
* right- No floating elements allowed on the right side
* both - No floating elements allowed on either the left or the right side
* inherit - The element inherits the clear value of its parent

When clearing floats, you should match the clear to the float: If an element is floated to the left, then you should clear to the left. Your floated element will continue to float, but the cleared element will appear below it on the web page.

Look at: 03-Layout/02-Float2

If an element is taller than the element containing it, and it is floated, it will "overflow" outside of its container. We can add **overflow: auto**; to the containing element to fix this problem. The **overflow: auto** clearfix works well as long as you are able to keep control of your margins and padding (else you might see scrollbars). The new, modern clearfix hack however, is safer to use, and the following code is used for most webpages:

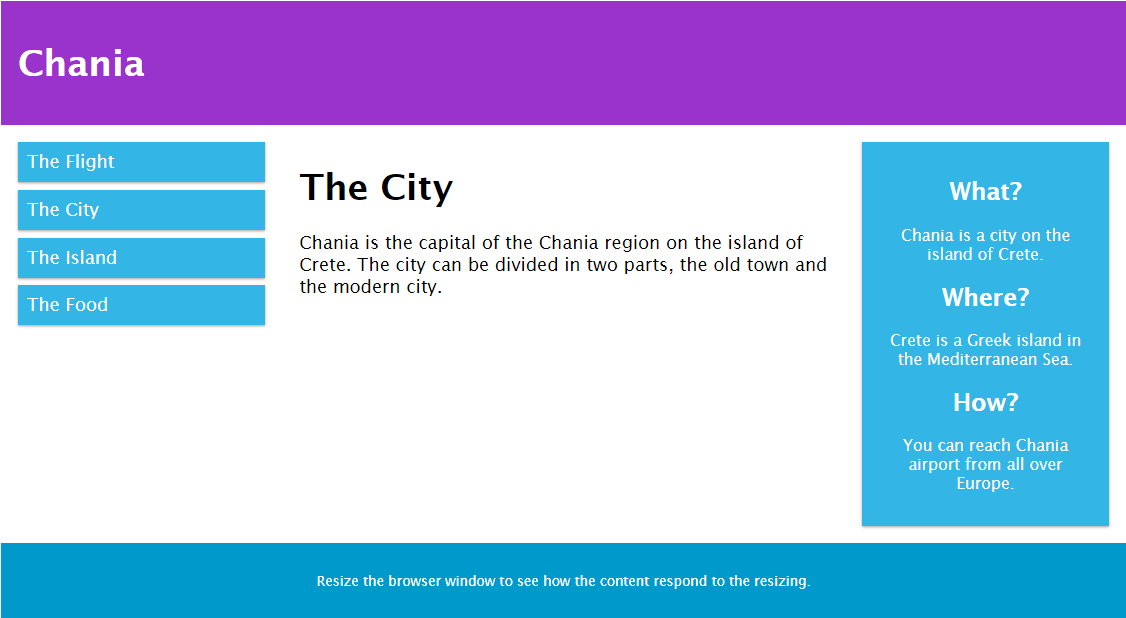
|  |
| --- |
| .clearfix::after {  content: "";  clear: both;  display: table;  } |

Look at: 03-Layout/03-ClearFix/index1.html

Look at: 03-Layout/03-ClearFix/index2.html

**Web Layout Example**

Now that we know how to place multiple elements on the same line side-by-side using the **float** property, we will design a simple Web page layout. Web pages with the following layout are common in practice:



Here we see that the page consists of a header and a footer, both of which take up the entire line. The heart of the web page consists of a container with three elements: A navigation bar that occupies about 25% of the space and floats to the left, a content area in the middle that occupies 50% of the space, and an aside that occupies about 25% of the space and is on the right. Here is the CSS and HTML for this page:

Look at: 03-Layout/04-ExWebLayout

One thing to notice in this example is that at the beginning of the CSS file, we set **box-sizing*:*** *border-box*; for all HTML elements in the page. The CSS **box-sizing** property allows us to include the padding and border in an element's total width and height.

Recall that by default, the width and height of an element is calculated like this:

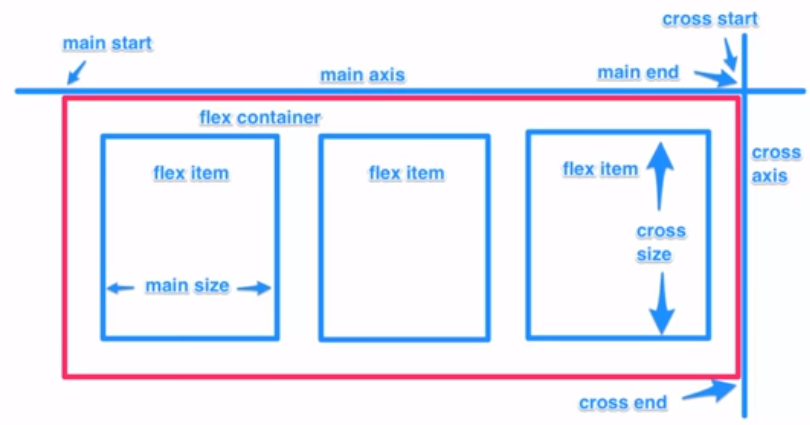
* width + padding + border = actual width of an element
* height + padding + border = actual height of an element

This means that when you set the width/height of an element, the element often appears bigger than you have set (because the element's border and padding are added to the element's specified width/height). The box-sizing property solves this problem by allowing us to include the padding and border in an element's total width and height. Thus, all elements that have the same width and height will indeed have the same width and height even if they have different margins and padding. Since the result of using the **box-sizing:** *border-box*; is so much better, many developers want all elements on their pages to work this way.

**Flexbox [**<https://www.w3schools.com/css/css3_flexbox.asp>**]**

As we have seen in the previous section, it is possible to place HTML elements side-by-side using the **float** property, but it has problems: Firstly, you have to clear the float property before you start the next row, which is cumbersome. Secondly, it is difficult to make the height of all elements in a container equal. Thirdly, if you want to change the order in which the elements appear inside the container, you always have to change the HTML document and rewrite the order of the elements. It is not possible to change the order from within the CSS without changing the HTML document.

The CSS **flexbox** model was developed to solve all of the above problems. After defining a flex container, you can define the direction of the elements (**flex-direction**: *row* or *column*), whether the elements would wrap to the next row if necessary (**flex-wrap**: *wrap*), where they should be aligned horizontally (**justify-content**: *flex-start*, *flex-end*, or *center.* default*: stretch*) or vertically inside the container (**align-items**: *flex-start*, *flex-end*, or *center*) etc.



As shown above, you define a flex-container and assign its **display** property to be *flex*. You then add flex items as child elements of this container. By changing different flexbox properties, you determine how each flex item gets displayed within the container. Here is a list of flexbox properties:

|  |  |  |
| --- | --- | --- |
| display | flex | inline-flex |  |
| flex-direction | row | column |  |
| flex-wrap | wrap | nowrap | wrapreverse |  |
| flex-basis | <length> | The same as the width property, e.g., 25% |
| justify-content | flex-start | flex-end | center | space-between | space-around | space-evenly | align the items along the main axis |
| align-self | flex-start | flex-end | center |  |
| align-items | flex-start | flex-end | center | stretch | baseline | align the items along the cross axis |
| align-content | flex-start | flex-end | center |  |
| flex-grow | <number> |  |
| flex-shrink | <number> |  |
| flex | <integer> | width of an item within the container in relation to other items |
| order | <integer> |  |

Look at: 03-Layout/05-Flexbox/index1.html

To see how 3 flex-items (boxes) are displayed within a flex-container, open the following example and “inspect” the page by pressing F12 in Chrome. Then select the “container” div. You will see a “flex” sign on the right. Press on it and different flexbox properties will appear. You can change these properties to see how the flex-items are displayed within the flex-box. Here, especially play with **justify-content** property, which is used to align the items on the main axis, and **align-items**, which is used to align the items on the cross axis.

Look at: 03-Layout/05-Flexbox/index2.html

Here is another example that contains different flexbox items within flex box containers:

Look at: 03-Layout/05-Flexbox/index3.html

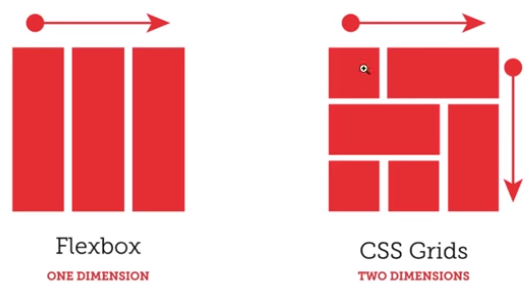
Here is a page where you can play with different flex properties: <https://codepen.io/enxaneta/full/adLPwv/>

Here are some CSS Flexbox Tutorials:

1. Net Ninja: <https://www.youtube.com/playlist?list=PL4cUxeGkcC9i3FXJSUfmsNOx8E7u6UuhG>
2. Brad Traversy: <https://www.youtube.com/watch?v=3YW65K6LcIA>

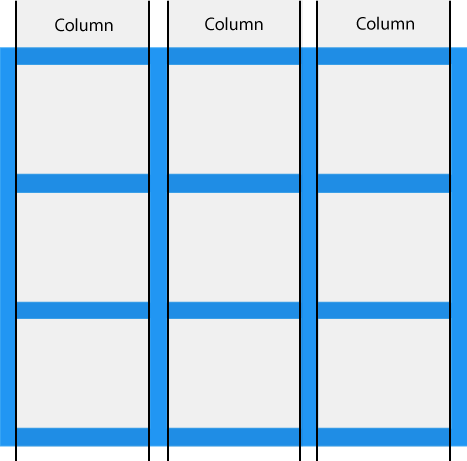
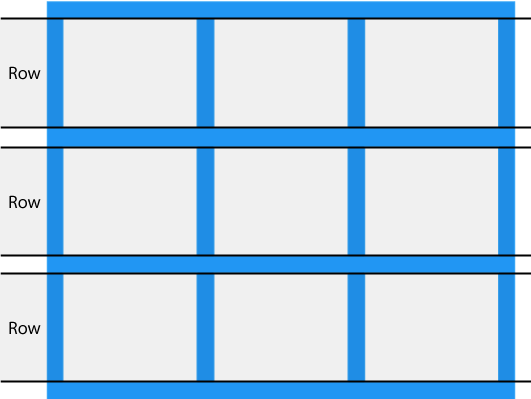
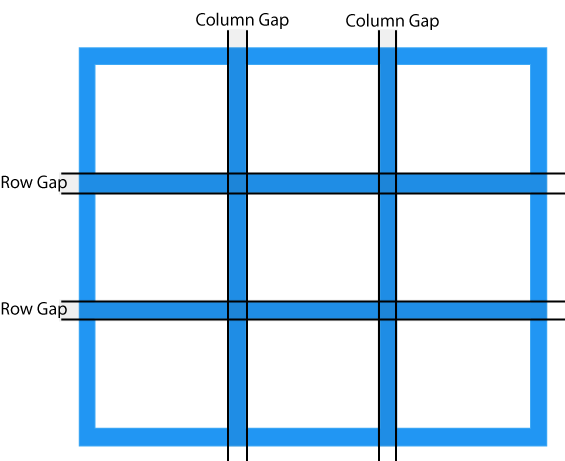
**CSS Grid [**<https://www.w3schools.com/css/css_grid.asp>**]**

The CSS Grid Layout offers a grid-based layout system, with rows and columns, making it easier to design complex web pages. While the flexbox is good for one-dimensional layouts, grid allows two-dimensional layouts to be created as shown below:



You can obviously emulate the two-dimensional layout shown above with multiple flexbox containers and elements, but it will be a lot more complicated. With CSS Grid, you can create the same layout within a SINGLE container, which makes creating sort of any layout very simple. So, if you need complex layouts, use CSS Grid. Otherwise, use flexbox.

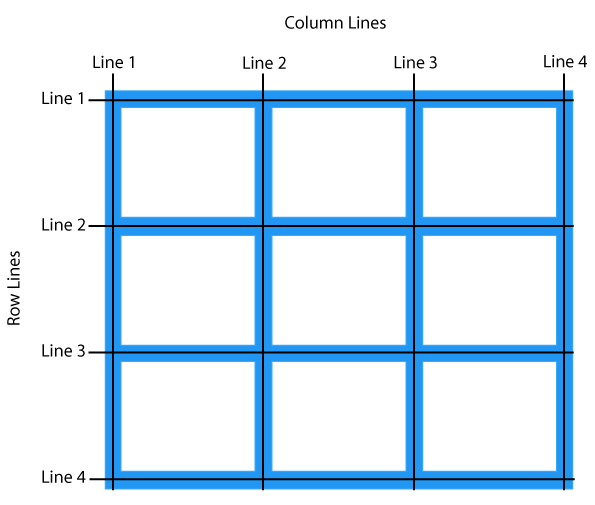
As the name implies, CSS Grid follows a two-dimensional grid system as shown below:

You can adjust the gap size by using one of the following properties: grid-column-gap, grid-row-gap, grid-gap. The **grid-gap** property is a shorthand property for the grid-column-gap and the grid-row-gap properties.

|  |
| --- |
| .grid-container {  display: grid;  grid-row-gap: 50px;  grid-column-gap: 40px;  /\* grid-gap: 50px \*/  } |

The following figure shows the grid lines.



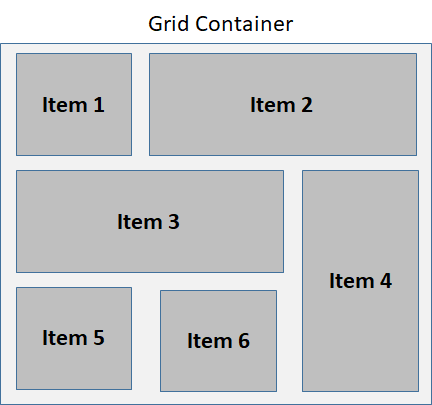
Place a grid item at column line 1, and let it end on column line 3 at row 1.

|  |
| --- |
| .item1 {  grid-row: 1;  grid-column-start: 1;  grid-column-end: 3;  } |

Place a grid item at column 2, from row 2 to 4:

|  |
| --- |
| .item2 {  grid-column: 2;  grid-row-start: 2;  grid-row-end: 4;  } |

This way, we can precisely place all items inside the grid-container. Let’s now create the following grid container and place the items as shown in the following figure:



Look at: 03-Layout/06-Grid/index1.html

The **grid-template-columns** property defines the number of columns in your grid layout, and it can define the width of each column. The value is a space-separated-list, where each value defines the length of the respective column. If you want your grid layout to contain 4 columns, specify the width of the 4 columns (fixed size in pixels or in percentages), or "auto" if all columns should have the same width. Here is a container that consists of 4 columns of equal size.

|  |
| --- |
| .grid-container {  display: grid;  grid-template-columns: auto auto auto auto;  } |

Here is an example where we set a size for the 4 columns:

|  |
| --- |
| .grid-container {  display: grid;  grid-template-columns: 80px 20% auto 40%;  } |

Look at: 03-Layout/06-Grid/index2.html

The **grid-template-rows** property defines the height of each row.

The **justify-content** property is used to align the whole grid inside the container. Possible values for the property are *space-evenly, space-around, space-between, center, start, end, stretch*. The grid's total width has to be less than the container's width for the justify-content property to have any effect.

The **align-content** property is used to *vertically* align the whole grid inside the container. Possible values for the property are *space-evenly, space-around, space-between, center, start, end, stretch*. The grid's total height has to be less than the container's height for the align-content property to have any effect.

Look at: 03-Layout/06-Grid/index3.html

Instead of using **grid-column-start** and **grid-column-end** to specify the start and end of a column, we can simply use **grid-column**: *start/end* to specify the same thing. This applies to **grid-row-start** and **grid-row-end** properties.

Look at: 03-Layout/06-Grid/index4.html

The **grid-area** property can be used as a shorthand property for the grid-row-start, grid-column-start, grid-row-end and the grid-column-end properties. Here is “item8" starts on row-line 1 and column-line 2, and end on row-line 5 and column line 6:

|  |
| --- |
| .item8 {  grid-area: 1 / 2 / 5 / 6;  } |

It is also possible to name all items, and make a ready-to-use webpage template:

|  |
| --- |
| .item1 { grid-area: header; }  .item2 { grid-area: menu; }  .item3 { grid-area: main; }  .item4 { grid-area: right; }  .item5 { grid-area: footer; }  .grid-container {  grid-template-areas:  'header header header header header header'  'menu main main main right right'  'menu footer footer footer footer footer';  } |

Look at: 03-Layout/06-Grid/index5.html

Here are some CSS Grid Tutorials:

1. Net Ninja: <https://www.youtube.com/playlist?list=PL4cUxeGkcC9hk02lFb6EkdXF2DYGl4Gg4>
2. Brad Traversy: <https://www.youtube.com/watch?v=0xMQfnTU6oo>

**Responsive Web Page Design [**<https://www.w3schools.com/css/css_rwd_intro.asp>**]**

*Responsive Web Design* is about using HTML and CSS to automatically resize, hide, shrink, or enlarge a web page to make it look good on all devices (desktops, tablets, and phones). For example, consider our example web page layout from the previous section. Although we want the navigation bar, the content and the aside to be side-by-side on a large screen, it might be better to stack the aside vertically on a table, and the navigation bar, the content and aside vertically one after the other for mobile devices with a small screen. Web pages that adjust their layout based on the size of the screen they get displayed are called *responsive web pages*. Here is how you might want your page to adjust based on the screen size:

**Desktop view Tablet view Phone view**

**Viewport**

The viewport is the user's visible area of a web page. The viewport varies with the device, and will be smaller on a mobile phone than on a computer screen. Before tablets and mobile phones, web pages were designed only for computer screens, and it was common for web pages to have a static design and a fixed size. Since the web page had a single view for all devices, the page appeared terribly on devices with smaller screens.

To solve this problem, HTML5 introduced a method to let web designers take control over the viewport, through the <meta> tag. When making *responsive web pages*, add the following <meta> element in the head section of your Web page:

|  |
| --- |
| <meta name="viewport" content="width=device-width, initial-scale=1.0"> |

A <meta> **viewport** element gives the browser instructions on how to control the page's dimensions and scaling. The *width=device-width* part sets the width of the page to follow the screen-width of the device (which will vary depending on the device). The *initial-scale=1.0* part sets the initial zoom level when the page is first loaded by the browser.

**Size Content to the Viewport**

Users are used to scrolling websites vertically on both desktop and mobile devices - but not horizontally! So, if the user is forced to scroll horizontally, or zoom out, to see the whole web page it results in a poor user experience. Some additional rules to follow:

1. Do NOT use large fixed width elements - For example, if an image is displayed at a width wider than the viewport it can cause the viewport to scroll horizontally. Remember to adjust this content to fit within the width of the viewport.
2. Do NOT let the content rely on a particular viewport width to render well - Since screen dimensions and width in CSS pixels vary widely between devices, content should not rely on a particular viewport width to render well.
3. Use *CSS media queries* to apply different styling for small and large screens - Setting large absolute CSS widths for page elements will cause the element to be too wide for the viewport on a smaller device. Instead, consider using relative width values, such as **width: 100%**. Also, be careful of using large absolute positioning values. It may cause the element to fall outside the viewport on small devices.

**Media-Queries [**<https://www.w3schools.com/cssref/css3_pr_mediaquery.asp>**]**

Media query is a CSS technique introduced in CSS3 to allow you write CSS rules that gets applied when the screen size is smaller than or bigger than a certain value. This way, you can change the layout of your Web page depending on the screen size of the device. It uses the @media rule to include a block of CSS properties only if a certain condition is true.

Two of the most-widely used media query properties are **min-width** and **max-width**.

In the following example, the background-color is blue for screen sizes > 600 and red for screen size <= 600. Notice that the default rule body {background-color: blue;} is specified for desktops and tablets. The special rule inside the media query kicks in when the screen size <= 600px.

|  |
| --- |
| body {  background-color: blue;  }  /\* The rule inside the media query becomes active when the screen-size <= 600 px \*/  @media screen and (**max-width**: 600px) {  body {  background-color: red;  }  } |

Look at: 03-Layout/07-ResponsiveDesign/index1.html

In the following example however, the background-color is blue for screen sizes < 600 and red for screen size >= 600. Notice that the default rule body {background-color: blue;} is specified for mobile devices. The special rule inside the media query kicks in when the screen size >= 600px. This sometimes is called *mobile-first approach*, where you design the page for mobile devices and then put media queries that change the appearance of the page for tablets and desktops. But mobile devices have priority in design over tablets and desktops.

|  |
| --- |
| body {  background-color: blue;  }  /\* The rule inside the media query becomes active when the screen-size >= 600 px \*/  @media screen and (**min-width**: 600px) {  body {  background-color: red;  }  } |

Look at: 03-Layout/07-ResponsiveDesign/index2.html

There are tons of screens and devices with different heights and widths, so it is hard to create an exact breakpoint for each device. To keep things simple you could target five groups:

|  |
| --- |
| /\* Extra small devices (phones, 600px and down) \*/ @media only screen and (max-width: 600px) {...}  /\* Small devices (portrait tablets and large phones, 600px and up) \*/ @media only screen and (min-width: 600px) {...}  /\* Medium devices (landscape tablets, 768px and up) \*/ @media only screen and (min-width: 768px) {...}  /\* Large devices (laptops/desktops, 992px and up) \*/ @media only screen and (min-width: 992px) {...}  /\* Extra large devices (large laptops and desktops, 1200px and up) \*/ @media only screen and (min-width: 1200px) {...} |

**Orientation: Portrait / Landscape**

Media queries can also be used to change layout of a page depending on the orientation of the browser. You can have a set of CSS properties that will only apply when the browser window is wider than its height, a so called "Landscape" orientation:

|  |
| --- |
| @media only screen and (orientation: landscape) {  body {  background-color: lightblue;  }  } |

**Example Responsive Web Page**

Let’s now redesign our example web page for 3 different page sizes:

* For screen sizes up to 600px (mobile), we want the menu, the content and the aside bar to appear vertically-stacked.
* For screen sizes between 600px-768px (tablet), we want the menu and the content side-by-side and aside to be down below them.
* The screen sizes > 768px (desktop, which is the default), we want the menu, the content and the aside to appear side-by-side.

Look at: 03-Layout/07-ResponsiveDesign/index3.html

Here is the same example Web page, but this time *mobile-first*, i.e., the default values are set up for the mobile device size, and @media queries have been added to change the layout for tablet and desktop.

Look at: 03-Layout/07-ResponsiveDesign/index4.html

The above examples used the float property for page layout. The next example shows the same responsive Web page designed using CSS flexbox. Notice how small the change is from float to flexbox.

Look at: 03-Layout/07-ResponsiveDesign/index5.html

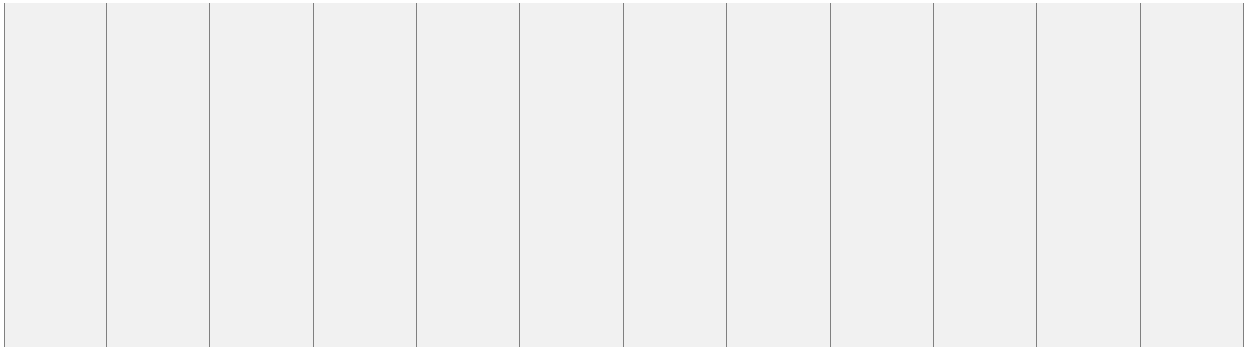
The final example shows the same responsive Web page designed using CSS Grid.

Look at: 03-Layout/07-ResponsiveDesign/index6.html (Using fixed-size fractional columns)

Look at: 03-Layout/07-ResponsiveDesign/index7.html (Using auto-sized columns)

**Grid-View**

Many web pages are based on a grid-view, which means that the page is divided into columns:



Using a grid-view is very helpful when designing web pages. It makes it easier to place elements on the page.



A responsive grid-view often has 12 columns, and has a total width of 100%, and will shrink and expand as you resize the browser window. The way such a grid system is implemented is to define 12 classes corresponding to different number of columns. Since we have 12 columns, 1 column occupies 100/12 = 8.33% of the container.

|  |
| --- |
| .col-1 {width: 8.33%;}  .col-2 {width: 16.66%;}  .col-3 {width: 25%;}  .col-4 {width: 33.33%;}  .col-5 {width: 41.66%;}  .col-6 {width: 50%;}  .col-7 {width: 58.33%;}  .col-8 {width: 66.66%;}  .col-9 {width: 75%;}  .col-10 {width: 83.33%;}  .col-11 {width: 91.66%;}  .col-12 {width: 100%;} |

Given these classes, we simply define how much space each element inside the container needs to occupy for different screen sizes, and you have a responsive web page. Here is an example with flexbox:

Look at: 03-Layout/08-GridSystem/index1.html

When you look at this example you notice that we define classes with different names for different screen sizes. For example, for mobile phones, the name of the class is col-xs-6. For tables, it is col-s-6, and for desktop, it is col-6. We define these classes within media queries designed for each device type. The order in which the media queries are placed inside the style file is important. For the system to work correctly, you must place the media query for the mobile phones first, followed by the media query for the tablets and finally the media query for the desktop. The reason is obvious: As the screen size grows, you want the class for the particular screen size to take over, overriding the styles defined in the previous class declaration, which is for the smaller screen size. Thus when you have an element having classes: "col-xs-12 col-s-6 col-3", this would really mean that you want the element to occupy:

* 12 columns (100%) of its container on a mobile phone
* 6 columns (50%) of its container on a tablet
* 3 columns (25%) of its container on a desktop

Since we want all elements to be 100% of the screen width in mobile phones, we do not even have to specify col-xs-12 at all. It is the default as seen in the following example. Also notice that Notice also that we used CSS flexbox with wrap enabled for the container element.

Look at: 03-Layout/08-GridSystem/index2.html

As we will see next week, this is exactly how **bootstrap** works: By default all elements are assumed to be displayed on a mobile screen at full height. If you want to change the layout as the screen gets bigger, you need to specify how much space they need to occupy at each break point as we illustrated in our examples. This is why such design is called mobile-first designs.

**More Resources to learn flexbox, grid, CSS animation**

Here is a video by Brad Traversy, where he builds a Responsive Website using HTML, CSS Grid, Flexbox and CSS animations. It is a great video to put everything in place: <https://www.youtube.com/watch?v=p0bGHP-PXD4>