**CSS**

1. Box model:

* Sizing
* Intrinsic sizing:
* Asking the content how much space it's going to take.
* Dynamic box.
* Size of the content changes, so changes the box size.
* It can be achieved by either unsetting the width or setting the width as min-content. (min-content tells the box to be as wide as the minimum width of its content)
* Extrinsic sizing:
* Box is predefined and we have to fit our content into that box.
* Explicitly setting the content size (there’s a limit of how much content you can add before it overflows out of the box’s bounds)
* Areas of box model:

Margin Box

Content Box

Padding Box

Border Box

* Background image is visible in padding box region.
* Outline and box-shadow is present outside the box and occupies in the margin box region.
* Every browser applies a user agent stylesheet to HTML documents. They provide sensible defaults to make content easier to read. They define how elements should look and behave if there's no CSS defined. It contains
* **Display property:** For <div> it’s block, <span> it’s inline, <li> it’s list-item (Difference between inline and inline block is we can’t specify height and width in case of inline but we can specify in inline-block and margin and padding can also be applied in inline-block which is not possible in inline)
* **Box-sizing property:** It tells the box how to calculate it’s box size. It has two values. One is default content-box and another is border-box. In content-box the padding and border will be appended to the width and height of the content box. But in border-box the padding and border will be included inside the height and width of the content box.

1. **Selectors:**

* To apply CSS to an element you need to select it
* The parts of a CSS rule: block of code containing one or more selectors and one or more declarations (property and value pairs).
* Simple selectors:
* **Universal selectors (\*)**:
* It selects every HTML element.
* Also called as wildcard.
* **Type selectors:**
* It matches to a HTML element directly.
* Ex: section, h1 etc.
* **Class selectors:**
* It matches any element that has that class applied to it.
* Uses a ‘.’ symbol before it to instruct the CSS language to match with only the class attributes.
* We can’t start the class name with a number.
* **Id selectors:**
* It matches the only element that has the id applied to it.
* Used a ‘#’ symbol before it to instruct CSS language to match with only the id attributes.
* **Attribute selectors:**
* It matches elements that have a certain HTML attribute, or have a certain value for an HTML attribute.
* These are identified by wrapping the selectors in a square bracket (‘[]’).
* Ex: [data-type='primary'] {

color: red;

}

* We can make the matches case-sensitive by adding a ‘s’ operator to our attribute selector or make it insensitive by adding ‘i’ operator to the attribute selector.

Ex: [data-type=’primary’ s] or [data-type=’primary’ i]

* We can also match portion of strings inside attribute values.

Ex:

/\* A href that contains "example.com" \*/  
[href\*='example.com'] {  
 color: red;  
}  
  
/\* A href that starts with https \*/  
[href^='https'] {  
 color: green;  
}  
  
/\* A href that ends with .com \*/  
[href$='.com'] {  
 color: blue;  
}

* **Grouping selectors:**
* A selector doesn’t need to match with only a single element. We can group multiple selectors by combining them with a comma.
* Ex:

strong,  
em,  
.my-class,  
[lang] {  
 color: red;  
}

* **Pseudo classes:**
* It selects HTML elements in different states.
* Like hover, nth-child etc.
* These are used with a single colon (‘:’) icon.
* **Pseudo element:**
* It is used to select part of an element or inserting to an element(using ‘content’ keyword).
* Like before, after, marker, selection etc.
* These are used with double colon (‘::’) icon.
* Complex selectors:
* Cascade downward selection
* 4 types of combinators:
* **Descendants combinator (space)**: all direct or indirect child elements
* **Child combinator (>):** only direct child elements
* **Next sibling combinator (+):** only first immediate next sibling that shares the common parent
* **Subsequent sibling combinator (~):** all siblings that shares the same parent
* Compound selectors:
* Ex:

a.my-class {  
 color: red;  
}

* Used without any combinators in between
* Only selects the <a> element that also contains the class my-class

1. The cascade:

* The cascade is the algorithm for solving conflicts where multiple CSS rules apply to an HTML element.
* Cascade algorithm has four different stages.
* **Position and order of appearance:**
* **Specificity:**

id > class > div

* **Origin:**

user agent base style (browser applies to HTML elements by default) < local user styles (come from the operating system level, browser extensions) < authored CSS (The CSS that you author) < authored !important < local user styles !important < user agent !important

* **Importance:**

Font-size, background, color < animation < !important < transition

1. Specificity:

* Specificity scoring:
* Universal selector: 0 point
* Element (Type), Pseudo-element selector: 1 point
* Class, pseudo-class, attribute selector: 10 points
* Id selector: 100 points
* Inline CSS: 1000 points
* !important rule: 10,000 points
* If specificity same, then newer CSS will be applied.

1. Inheritance:

* Some CSS properties inherit if you don't specify a value for them.
* The root element (<html>) won't inherit anything because it is the first element in the document. Add some CSS on the HTML element, and it starts to cascade down the document.
* Some inheritable CSS properties are: color, cursor, font-family, font-size, font-style etc.
* Every HTML element has every CSS property defined by default with an initial value.
* If no explicit value for a property is mentioned, then if the property is inheritable, it would inherit the computed value from its parents otherwise stick with the initial value of the element.
* Explicit inheritance using ‘inherit’ keyword (for both inheritable and non-inheritable properties).
* We can reset the property back to its initial default value using ‘initial’ keyword.
* We can select all the property of an element using ‘all’ keyword.
* ‘Unset’ keyword sets the property to inherit if the property is inheritable otherwise keep it initial.

1. Color:

* Red, green and blue are primary colors.
* Numerical colors:
* Hex colors:
* Syntax:

h1 {

color: #b71540;

}

* According to the Web Almanac, hex is the most popular color syntax type.
* The hexadecimal ranges are 0-9 and A-F with 6-digit sequence which are translated to RGB numerical ranges i.e., 0 - 255.
* You can also define an alpha value with any numerical colors. An alpha value is a percentage of transparency.
* In hex code, you add another two digits to the six-digit sequence, making an eight-digit sequence.
* 0% alpha (Fully transparent) – 00

50% alpha – 80

75% alpha – BF

* We can write 6-digit hex code in a 3-digit format. Like #aa44ee can be written as #a4e in short form. And we can also add alpha value to it like #a4e8 which is equivalent to #aa44ee88.
* We can also convert the alpha value to the equivalent decimal value by multiplying it with its base i.e., 16 and it ranges between 0 – 255.
* RGB colors:
* It is represented with rgb() color function.
* It takes number or percentage as parameter.
* The numbers need to be within the 0-255 range and the percentages are between 0% and 100%‌. RGB works on the 0-255 scale, so 255 would be equivalent to 100%, and 0 to 0%.
* Black – rgb(0, 0, 0) or rgb(0%, 0%, 0%)

White – rgb(255, 255, 255) or rgb(100%, 100%, 100%)

* We can mention alpha in two ways by either writing the alpha value after giving a ‘/’ in the rgb function or by writing rgba() function.
* Alpha ranges between 0 – 1. 50% transparency means either 0.5 in numerical or 50% in percentage.
* Commas were removed from the rgb() function because spaces are being used as delimiter.
* HSL (Hue, saturation, lightness):
* Hue describes the value on the color wheel, from 0 to 360 degrees, starting with red (being both 0 and 360).
* A hue of 180, or 50% would be in the blue range.
* Saturation is how vibrant the selected hue is.
* A fully desaturated color (with a saturation of 0%) will appear grayscale.
* And finally, lightness is the parameter which describes the scale from white to black of added light. A lightness of 100% will always give you white and 0% will always give you black.
* Hue accepts number type, degree, turn, radian and gradient.
* Syntax: For black: hsl(0deg 0% 0%) or hsl(0 0% 0%) or hsl(0turn 0% 0%) etc.
* Alpha is defined in hsl(), in the same way as rgb() by adding a / after the hue, saturation and lightness parameters or by using the hsla() function.
* Lab() and lch()
* Color Keyword:
* There are 148 named colors in CSS. These are plain English names such as purple, tomato and goldenrod.
* Some of the most popular names, according to the Web Almanac, are black, white, red, blue and gray. Our favorites include goldenrod, aliceblue, and hotpink.
* transparent is a fully transparent color. It is also the initial value of background-color.
* ‘currentColor’ keyword refers to the value of color property of an element.
* Use case of color data type are: color, text-shadow, text-decoration-color, background-color, border-color, outline-color, box-shadow, gradient(linear-gradient) etc.
* Gradient gives smooth transition between two colors.
* Gradient is an image which is programmatically defined in CSS.

1. Sizing Units:

* ‘ch’ unit allows to limit the width of a text with a unit whose width is the width of character ‘0’ at the computed size.
* Numbers:
* Numbers are unitless integers and decimals.
* A number is a ratio if present without any unit.
* Like in line-height 1.5 means 150% of the element’s computed pixeled font-size.
* When setting values for filters: filter: sepia(0.5) applies a 50% sepia filter to the element.
* When setting opacity: opacity: 0.5 applies a 50% opacity.
* In color channels: rgb(50, 50, 50), where the values 0-255 are acceptable to set a color value.
* To transform an element: transform: scale(1.2) scales the element by 120% of its initial size.
* Percentage:
* When width of an element is calculated as percentage, it’s size becomes the portion of available width of parent element.
* When padding and margin are calculated as percentage, they will be portion of the parent element’s width irrespective of any direction.
* When transform is calculated as percentage, it will be portion of the elements computed width.
* Dimensions and lengths:
* If we attach unit to a number, it becomes dimension.
* When dimensions are related to distance, it is called as length. It can be absolute or relative.
* Absolute values can’t change but relative values can change.
* Absolute units: cm, mm, px, in etc.
* Relative units:
* Font-size-relative-unit: em(relative to parent elements size), rem(relative to root element or document base size), ch, lh(line-height), rlh etc.
* View-port-relative-unit: vw(relative to view port width), vh(relative to view port height), vmin, vmax etc.
* Angle units: deg, rad(radian), grad(gradian), turn(1 turn = 360degree)
* Resolution units: dpi(dots per inch)

1. Layout:

* Uses ‘display’ property.
* Two main layout mechanism is there to create layout. Those are flexbox, grid.
* Flexbox:
* One-dimensional layout mechanism (across one axis either horizontally or vertically)
* Syntax: display: flex
* makes the box a block-level box, and also converts its children to flex items.
* By default, flexbox will align the element's children next to each other, in the inline direction, and stretch them in the block direction, so they're all the same height.
* Items will stay on the same axis and not wrap when they run out of space (instead the height will increase) which can be changed using align-items, justify-content and flex-wrap properties.
* You can also change how it shrinks or grows using the flex property.
* The flex property is a shorthand for flex-grow, flex-shrink and flex-basis.
* Grid:
* It is designed to control multi-axis layouts instead of single-axis layouts (vertical or horizontal space).
* Syntax: display: grid
* It uses few new functions like repeat(), minmax() and a unit fr (fraction of remaining spaces).
* We could define that the first item in this grid takes up 2 rows and 3 columns by using grid-row and grid-column properties.
* The grid-row and grid-column properties instruct the first element in the grid to the start from the first column till of the fourth column, then span to the third row, from the first row.
* Inline-block:
* Using inline-block gives you a box that has some of the characteristics of a block-level element, but still flows inline with the text.
* Padding, margin, height and width will work.
* Float:
* instructs an element to "float" to the direction specified, which then allows sibling elements to "wrap" around it.
* Can be left, right or inherit.
* Better to add “display: flow-root” to avoid the page disruption.
* Multicolumn:
* When we have a long list item, then instead of endless scrolling we can use multicolumn to split the list into multiple columns.
* It can be done in two ways by using column-count property or setting explicitly the width of column:
* .countries {

column-count: 2;

column-gap: 1em;

}

* .countries {

width: 100%;

column-width: 260px;

column-gap: 1em;

}

* Positioning:
* The position property changes how an element behaves in the normal flow of the document, and how it relates to other elements.
* It can be relative, absolute, fixed and sticky with the default value being static.
* Relative position is positioned relative to its normal position.
* Absolute position is positioned relative to the nearest positioned ancestor.
* Fixed position is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled.
* Sticky position is positioned based on the user's scroll position.
* A sticky element toggles between relative and fixed, depending on the scroll position. It is positioned relative until a given offset position is met in the viewport - then it "sticks" in place (like position: fixed)

1. Flexbox:

* There are two possible axes. One is main-axis and another one is cross-axis.
* Main-axis is set by the ‘flex-direction’ property (default is row).
* The cross axis runs in the other direction to the main axis.
* Creating a flex container:
* Do this by changing the value of the display property to flex.
* This will give you a block-level box, with flex item children. The flex items immediately start exhibiting some flexbox behavior, using their **initial values**.
* The initial values mean that:
* Items display as a row.
* They do not wrap.
* They do not grow to fill the container.
* They line up at the start of the container.
* Value of flex-direction can be row, row-reverse, column and column-reverse.
* Reversing the flow of items and accessibility:
* We should be cautious when using any properties that reorder the visual display away from how things are ordered in the HTML document, as it can negatively impact accessibility.
* Ex: row-reverse, column-reverse
* Flex items lay out as a row by default. A row runs in the direction that sentences flow in your writing mode and script direction.
* The start of the main axis is referred to as main-start. So, our flex items initially line up from main-start. The end of that axis is main-end. The start of the cross axis is cross-start and the end cross-end.
* Wrapping flex items:
* The initial value of the flex-wrap property is ‘nowrap’. This means that if there is not enough space in the container the items will overflow.
* Items displaying using the initial values will shrink as small as they can, down to the min-content size before overflow happens.
* To cause the items to wrap add ‘flex-wrap: wrap’ to the flex container.
* When a flex container wraps it creates multiple flex lines. In terms of space distribution, each line acts like a new flex container.
* When we are in row 2, we can’t add items of row 1 in row 2, as it allows only to use it in one-dimension.
* Controlling space inside flex items:
* Uses the property ‘flex’ property.
* It stands for flex-grow, flex-shrink, flex-basis.
* It can have values like initial, auto, 1 and none.
* Initial:
* Flex-grow = 0
* Flex-shrink = 1
* Flex-basis = auto
* Auto:
* Flex-grow = 1
* Flex-shrink = 1
* Flex-basis = auto
* 1:
* Flex-grow = 1
* Flex-shrink = 1
* Flex-basis = 0
* None (inflexible behavior):
* Flex-grow = 0
* Flex-shrink = 0
* Flex-basis = auto
* We can also set different ‘flex’ properties for the different flex items. Like flex: 1, flex: 2, flex: 3. Here the entire space will be divided into 6 equal parts. One part will be given to 1st child, 2 parts will be given to 2nd child and 3 parts will be given to 3rd child.
* Flex: 2 means flex-grow = 2, flex-shrink = 1 and flex-basis = auto
* We can reorder flex-items using the order property. Items with less order value will appear first. If two items have same order value, both items will appear adjacent to each other, but the one placed first in the HTML document will appear first.
* Justify-content:
* It distributes the spaces on the main axis
* It can have values like flex-start(default), flex-end, center, space-between, space-around, space-evenly.
* Align-content:
* It distributes the spaces on the cross axis.
* It can have values like stretch(default), flex-start, flex-end, center, space-between, space-around, space-evenly.
* In all the cases the flex-items will always start from main-start
* Place-content:
* It can be used to define both justify-content and align-content.
* If one value is provided, sets the same value for both properties.
* If 2 values are provided, sets the first one for align-content and second one for justify-content.
* Align-self:
* It is used to align one flex item in cross-line.
* It can have values like stretch(default), flex-start, flex-end, center.
* Align-items:
* Used to align all the flex items as a group in cross line.
* It can have values like stretch(default), flex-start, flex-end, center.
* In flex-box there is no justify-self and justify-items as the items are considered as a group in the main axis.
* We can separate a particular item in the flexbox by giving it a margin of auto. Auto finds out the spaces in the direction of using it, and gives all the spaces as margin.

1. Grid:

* Grid lines – row lines, column lines
* Grid track – row track (space between two row lines), column track (space between two column lines)
* Grid cell – intersection of row and column tracks
* Grid area – several grid cells together
* Gaps – Gap between the tracks. We can’t put contents in these gaps but can span grid items across it.
* Grid container – HTML element where ‘display: grid’ is applied.
* Grid item – direct child of grid container
* We can set each row and column size in grid by using the property grid-template-rows and grid-template-columns. Auto in this case, sets the size to the size of the content.
* Grid tracks can use intrinsic sizing keywords like min-content, max-content, fit-content. Min-content is the minimum content size i.e., the width of the longest word in the content. In max-content the size is the size of the content if written in a single line. And in fit-content the size that has been passed in the parameter would be the max size for the tracks. Any track if has max-width less than the parameter then it’s size would be less than the parameter. If the content of the track is more than the parameter then the content will be wrapped.
* The fr unit:
* fr unit shares out available space
* It will share the space in proportion.
* Higher the fr value, gets more of the spared space.
* Minmax() function:
* Takes 2 parameters min and max.
* Repeat() function:
* It is used when the same track details we need to write for multiple times
* Syntax: repeat (no. of repetition, content to repeat)
* Autofill and autofit:
* Autofill automatically fills the rows with possible columns leaves the rest spaces as empty while autofit stretched the columns to fill the entire width of the grid.
* By default, the items are filled in grid in row wise. But we can fill the items column wise by setting the ‘grid-auto-flow: column’ or ‘writing-mode’ properties.
* We can span the grid-items to more than one track by setting ‘grid-column-end’ property of individual grid-items.

Syntax:

grid-column-end: span 2 // covering two tracks

Or grid-column: auto / span 2 /// auto will be treated as the value of grid-column-start and span 2 will be the value of grid-column-end.

* If there is any gap in the grid, we can fill the gap by setting the grid-auto-flow property to dense.
* We can place any item using the grid-row-start, grid-column-start, grid-row-end and grid-column-end properties or shortly, grid-row and grid-column properties.
* We can stack the items based on the z-index value. Higher the z-index, the element will be at the top.
* If we explicitly define the dimensions of grid-row and grid-column, it is known as explicit grid otherwise implicit grid.
* Negative line number refers to the end of the grid.
* Negative line numbering won’t work in cases of implicit grid.
* The tracks created in the implicit grid will be auto-sized by default.
* We can give names to grid lines which can be used to place items.

Syntax:

.container {  
 display: grid;  
 grid-template-columns:  
 [main-start aside-start] 1fr  
 [aside-end content-start] 2fr  
 [content-end main-end]; /\* a two column layout \*/  
}  
  
.sidebar {  
 grid-column: aside-start / aside-end;  
 /\* placed between line 1 and 2\*/  
}  
  
footer {  
 grid-column: main-start / main-end;  
 /\* right across the layout from line 1 to line 3\*/  
}

* We can give name to grid areas that can be used to place items.

Syntax:

header {  
 grid-area: header;  
}  
  
.sidebar {  
 grid-area: sidebar;  
}  
  
.content {  
 grid-area: content;  
}  
  
footer {  
 grid-area: footer;  
}

.container {  
 display: grid;  
 grid-template-columns: repeat(4,1fr);  
 grid-template-areas:  
 "header header header header"  
 "sidebar content content content"  
 "sidebar footer footer footer";  
}

* To leave space in the grid in case of grid-template-area, we can write ‘.’ Or multiple dots without any space.
* Grid-auto-rows and grid-auto-column creates implicit grid.
* The grid-template property is a shorthand for grid-template-rows, grid-template-columns and grid-template-areas.

First, we have to write grid-template-area, then row and column with a ‘/’ in between row and column.

* Shorthand properties: grid (grid-template-rows, grid-template-columns, grid-template-areas, grid-auto-rows, grid-auto-columns, grid-auto-flow), grid-template
* Alignment: justify-content, justify-self, align-content, align-self, justify-items, align-items

1. Logical properties:

* Block flow: flow of block elements from top to bottom
* Inline flow: flow of inline elements. In English from left to right, In Arabic from right to left
* Flow relative:
* Instead of using margin-top, we can use margin-block-start.
* Instead of max-height, min-height, we can use max-block-size, max-inline-size.
* Instead of text-align: right, we can use text-align: end
* Instead of top, bottom, left, right, we can use inset-block-start, inset-block-end, inset-inline-start, inset-inline-end
* Instead of border-bottom, border-bottom-right-radius, we can use border-block-end, border-end-end-radius.
* Two new logical units. Vi which is equivalent to vw and vb which is equivalent to vh.
* Browser support is required to write shorthand properties.

1. Spacing:

* Using HTML elements like <br> and <hr>
* Using margin property:
  + One value is for all sides, two value is for top/bottom and left/right. Three value is for top, left/right, bottom.
  + We can set the margin as auto, percentage, length.
  + We can also set negative values for margin. If the negative value is more than available space between the elements, then the elements will collide with each other.
  + Only the block margin collapse (not inline margin). Means it selects the largest value of two adjoining elements with vertical margin set on the adjoining sides.
  + If we use float or absolute property, then the margin won’t collapse.
  + In flex or grid, margin will combine instead of collapsing.
* Using padding property
* Using position property
* Values are static, relative, absolute, fixed, sticky.
* Static: default
* Relative: relative to the element’s position
* Absolute: relative to the parent’s position
* Fixed: relative to the viewport
* Sticky: depends on scrolling
* Using grid and flexbox:
* Using gap property shorthand property for row-gap and column-gap.
* It can take one or two values. One value means that will be applied to both row and column. Two value means first one is for row and second one is for column.
* Gap property takes values in lengths, percentage, inherit, initial, unset.
* Consistent spacing:
* Called as gutter
* We should save these values to custom CSS properties or variables, so that we can use it as a token throughout the CSS.
* Ex:

:root {

--gutter: 20px;

--spacing: 1em;

}

h1 {

margin-left: var(--gutter);

margin-top: var(--spacing);

}

* If the spacing is changes inside elements locally or globally, the values will cascade. And updated values will show.

1. Pseudo elements:

* A pseudo-element is like adding or targeting an extra element without having to add more HTML.
* ::first-letter is used to select the first letter of the element.
* Both the ::before and ::after pseudo-elements create a child element inside an element only if you define a content property.
* Content can be any string, empty string, image(unable to resize), counter()
* ::before and ::after only work with the elements those allow making child elements. Like in <video>, <img>, <input> tag it will not work. Exception input[type=”checkbox”] allows to have child elements.
* ::first-letter doesn’t allow all CSS properties. It allows only color, background (background-image), border(border-color), float, font (font-size, font-weight) and text properties (text-decoration, word-spacing).
* ::first-letter is only for block elements, not for inline.
* ::first-line
* Used to style first line of an element
* Only for those elements whose display value is block, inline-block, list-item, table-caption, table-cell.
* Only allows color, background, font, text CSS properties.
* ::back-drop
* Used to style the part between the view port and full screen mode video.
* Supported by every browser except safari.
* ::marker
* Used to style the bullet or number for a list item or the arrow of a <summary> element.
* Only allows color, content, white-space, font, animation, transition property.
* ::selection
* The ::selection pseudo-element allows you to style how selected text looks.
* Only allows color, background-color, text properties.
* ::placeholder
* Allows to style the placeholders.
* Only allows color, background, font, text CSS properties
* ::cue
* Used to style the captions of video elements.
* You can also pass a selector into a ::cue, which allows you to style specific elements inside a caption.
* Ex:

video::cue {

color: yellow;

}

video::cue(b) {

color: red;

}

video::cue(i) {

color: lightpink;

}

1. Pseudo-classes:

* Pseudo-classes let you apply CSS based on state changes. This means that your design can react to user input such as an invalid email address.
* :hover
* Used to style elements when they are hovered.
* :active
* Used to style elements when they are clicked before releasing.
* :focus
* Used to style elements when they are focused liked clicked or tabbed.
* :focus-within
* Used to style other child elements when a child element of an element is in focus.
* :focus-visible
* Used to style element when they are focused with the help of keyboard.
* :target
* If in a page there is an url with ‘#’ and id that is connected to a specific part of the same page, target pseudo class is used to design that targeted part
* :link
* The :link pseudo-class can be applied to any <a> element that has a href value that hasn't been visited yet.
* :visited
* Used to style the link which already has visited by the user.
* Color, background-color, border-color, outline-color and color of SVG fill and stroke are allowable CSS properties.
* Order matters in case of pseudo classes. Link can override visited with same specificity. So the order should be :link, :visited, :hover, :active
* :disabled and :enabled
* Disabled is used to style a disable button
* Enabled is used to style a normal button which is enabled by default.
* :checked and :interminate
* Checked is used to select the checked state of radio and checkbox
* Interminate is used to select the state in group of checkboxes where some of are checked and some of are not
* :placeholder-shown
* Used to style a state where there is a placeholder attribute but no value.