Basic CSS

* Case insensitive: use classes and id selectors like this: main-navbar
* Always use fallback fonts: serif, sans-serif
* Import fonts before importing the css main file
* How to select by attribute: [disabled] { color: red; }
* Chrome dev tools shows rules from more to less specific, (top to bottom)
* Inline styles are the more specific
* Bigger margin Wins
* Block level elements have width: 100% by default
* To set the high to 100% of an element with position: static | relative, the parent component (which has position: relative | static) has to have a fixed height: 100px; or a chain of height: 100% that goes up till the html element, because the parent has a dynamic height, to just contain the child, and this will lead to an infinite loop of calculating the height.
* If width: 100% applied to element with position: absolute, and there’s no parent with position property applied, the containing block is the viewport, like for fixed elements.
* \* {} targets every element in the DOM, but body {} just targets the body and normal inheritance is applied to child elements (and the browser default rules can ruin that inheritance, like box-sizing: border-box set to body, ruined by display: block set by the browser, which applies content-box again)
* active:not(p) {} selects every item with class active, excepts the paragraph. Not recommended to use.
* <a> items, which are inline, so they sit next to each other, have an inline box: any vertical padding, border, or margin applied to an element will not push away elements above or below it. To do so, apply display: inline-block
* Margin: auto, to center items in the x axe inside a box, like the circles of the project. The centered items should be block level elements with a width defined.
* To apply z-index, the element should have postion!= static (with a position property applied) or to a flex-items (even if they don’t have the position property)
* Position: fixed is relative to the view port, and absolute is relative to the html, or the closest parent which has a position property applied.
* If the parent element of a relative element is the body itself, I have to apply overflow:hidden to the body AND the html as well; because the body passes it’s overflow property to the html and loses it. So if both already have it, the body doesn’t need to pass it up, and hence, keeps it
* Nested fixed positions: the parent is the limit (it’s has a self stack context). If a have parent and child elements with position: fixed, if I give the child elements a z-index: 1000; it’s not absolute, it’s relative to the parent z-index, it’s just to order elements inside the parent, but for the outside world, what matters is the index of the parent.
* Background-size: 100px, sets the width and the height is auto, to keep the aspect-ratio
* If you have an image inside a container that’s an inline element (like an <a>), the image will take it’s default dimensions, no matter if you even set dimension of the container
* Box-shadow applied to images should add a vertical-align: top | bottom, to get rid of the white space at the bottom. Or set display: block to the image;
* Element with position: fixed and width:100%, takes the 100% of the vw
* If 2 elements have position: fixed, the second one on the hmtl code, will be on top of the first one. They’re out of the document flow, and have index: auto == 0
* Want to apply width to an element with %? Check out the 3 rules, and think about the properties of the current and parent elements
* Position: absolute, the anchor is the closest parent with position property applied, or the hmtl if no one is found
* Position: fixed: the anchor is the view port.
* If a have a parent with padding, it’s added outside the margin of the child

Parent with child, the bottom-margin is merged: <div>This parent element contains two paragraphs!

<p>This paragraph has a <code>.4rem</code> margin between it and the text above.</p>

<p>My bottom margin collapses with my parent, yielding a bottom margin of <code>2rem</code>.</p>

</div>

* <https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Box_Model/Mastering_margin_collapsing>
* Margin collapsing of header and ul inside it (the margin of the ul merges with the margin of the header).
* If I use the shorthand border: 2px solid red, and then add border-width:3px, I still keep the other properties. Warning! The other way round (border-width: 2px, then border: solid red; ) deletes all the properties previously set
* Elements can be outside of the parent’s box: e.g when the child has position: absolute and the overflow property of the parent is not set to hidden; or when there are many child and one of them claims the 100% of the height, so the other have to move out of the box ☹
* Display: none, doesn’t remove the element from the DOM, just from the visible document flow
* Inline-block elements take the whole available space, but if width: auto, then they behave like inline elements, and they take only the space they need.
* Inline elements, as they share the same line, they can’t decide themselves the width and height, and top and bottom margin and padding can’t be applied, but for inline-block elements is possible.

The line break between two inline elements is counted as a character and takes space:  <div>

            <a href="index.html" class="main-header\_\_brand">

                <img src="./images/uhost-icon.png" alt="uHost- Your favourite host site">

            </a>

        </div>

        <nav class="main-nav">

* Hence, to have the div and the nav in the same line, I use 30% of the width for the div and 65% for the nav, to give extra space for it.
* To align to inline elements, in the same line, apply vertical-align: middle;
* Pseudo elements, like :: after, are useful to add content after an element, for example Hello World! (extra text here).
* Always use classes, over id, even if you use that class once
* !important overrides specifity, so, even if there’s a more specific rule, the important wins
* To center inline elements (like text) , use text-align: center;
* Use float for laying out text and image, but not for block positioning, because it needs an extra div (clearfix)
* Background-size: contain shows the entire image, and there may be white spaces; but cover fills the container, but may not show the full image
* Background-position: center is the same as 50% 50% (of the x axe not displayed image, 50% is cropped at the bottom and the other half at the top, it’s like distributing the non-showed parts)
* Background image in a container with dotted border? Check out background-origin: border-box, to make the image be shown beneath left and right borders
* Want to exclude the image underneath the top and margin dotted borders? Check out background-clip: padding-box(not applicable to the traditional box-size)
* To size an <img>, you can: define the size of the image itself, or use percentages relative to the parent container (watch out for inline elements, and turn them into inline-block elements to define width and height).
* The default font-size of the html element (the root) is 16px
* Viewport: vmin and vmax are either the min or the max of the vh and the vw.
* Replace console.log by console.dir to see the object representation of a DOM element
* The styles key of the DOM object, shows the inline styles, not the ones applied through selectors.
* Element.style.backgroundColor is the same as Element.style[‘background-color’]

If a footer is displayed in the middle of the page because of a short main section, set the min-height that section, to be sure that the footer is always displayed at the bottom of the viewport

main {

    /\* min-height: calc(100vh - 3.5rem - 8rem); \*/

    margin-top: 3.5rem;

}

A solution for avoidin using cal() to deduct the footer and the navbar, is to use the grid system like this:

html {

    height: 100%;

}

body {

    font-family: 'Montserrat', sans-serif;

    margin: 0;

    display: grid;

    grid-template-rows: 3.5rem auto 8rem;

    /\* whith height defined, the auto should complete the full height, and not just the height required by the content \*/

    min-height: 100%;

}

main {

    margin-top: 3.5rem;

    grid-row: 2 / 3;

}

.main-footer {

    grid-row: 3 / 4;

}

The first row in not filled with any element, becausa it wans’t assgined to anyone, and the navbar has a position of fixed, so it’s not part of the grid

Responsive design

* If no scale is applied, a phone accommodate the almost the same content as a desktop, because the width in pixels of the device is like 720px, but in chrome dev tools shows 360px (the ‘css pixels), because it applies a pixel ratio of 2, in order to match the actual width of 3.75 inches of width.
* So, what is fixed is the width = 3.75inch, then converted to css pixels = 360px, that don’t match the phone pixel’s density = 720 density, which is fixed an read by the browser.
* So, it reads the 720px from the hardware, then divided by 2, to have 360px.
* Mobile first design: the rules are set to look good on mobile, and the media queries are set @media (min-width: xxxxx). Going from small to big.
* Media queries should follow this order in the css file: the smallest first
* To add an OR operator in media queries, just use a coma ,
* The landscape mode for desktop is reached when the width > height
* When setting media queries and looking for breakpoints, check for the computed width of elements if using border-box in the dev-tools

Forms

* Inputs and button, don’t take all the available space if turned into block elements, so width: 100% should be applied if I want so.
* Checkboxes don’t respond to changes in style applied to input elements, even when they are so.

Fonts and text

* it’s better to import the fonts in the css file, (like in shared.css) rather than in the html file.
* When importing fonts as .ttf files, add the font-weight to the imports in the css. And start the .tff file names with lowercase, for ie browser support. @font-face approach not recommended.
* Line-height defines the height of the content (e.g, in a <p>). If line-height: 2; (2xfont-size).
* Don’t use % for Line-height, because it can be applied to an inherited (not desired) font-size, and can cause problems
* Line-height doesn’t consider decimals, just round numbers
* The line-height: normal depends on the font-family used (‘Montserrat’, ‘Roboto’)
* Font shorthand: The font-size and font-family values are required. If one of the other values is missing, their default value are used.

Flexbox

* Flex container with display: flex; the block elements just take the width they need for the content, and if expanded, they reach the width specified in for each (like width: 300px). The width of the container changes if the dev tools bar moves. (taking the full width). If the width of the viewport isn’t enough for all the items, the parent shrinks and leaves items outside the box, on the right.
* display: inline-flex, doesn’t allow the block elements to shrink, it keeps their original width. The container behaves like an inline elements, because it only takes the space the items need, but not more, not less.
* The height of the containers lines are defined by the tallest item in each one
* Flex-wrap: wrap respects the width of the items, and sends them to a new line if they don’t fit
* Flex-wrap: wrap-reverse, piles up the non-fit elements on higher line, rather than below, like usual. And the items are aligned to the bottom.
* Flex-direction: column, the elements behave like block elements, take the full available width
* Flex-direction: row | column –reverse , the first element starts on the right or the bottom respectively.
* Flex-direction: row =>staring point is top left; row-reverse => top left
* Flex-direction: colum => starting point is same as above, but main and cross axis change
* Align-items aligns items in the cross axis
* Flex-direction: column, makes the top and bottom margin of items not collapsible;
* Instead of using display: inline-block to place elements in the same line, use display: flex in the container
* With order: -1; I can move a flex item to the beginning of the items. The default value is 0;
* Align-items applies (in the flex container) to all the items, but align-self applies just to one item ;
* By default, items don’t grow, so flex-grow: 0 by default;
* Flex-shrink: 0; makes the item nit shrinkable, and it keeps it defined width (if it has one). By default, items have flex-shrink: 1.
* Flex-basis refers to the Main axis.
* If flex-basis is set, it overrides the width propery for row direction, and height for column direction
* Flex-basis: auto will fallback to the set width or height values, depending on the main axis (flex-direction: row => width, column => height)
* Flex-basis: 10% makes the width equal to 10% of the flex container content (because it’s like a block level element)(see units section of the course)

CSS grid

* The fr unit distributes the remaining space according to the the fractions assigned.
* Column with width of auto, takes the remaining space, but for rows, it just takes the height the content needs (unless the height of the container is set.
* Using minmax for row height, the row it’s gonna take the maximum available (take into account restriction because of the parent’s height, if defined).
* To make an element span across all columns, grid-column-end: -1;
* Elements can overlap in the grid, if they’re forced to, with a combination of grid-row/column-start/end. Otherwise, they don’t do it. The one on top depends on the order in the dom (kind of the order in the html doc)
* The grid system supports z-index, even with not a position property applied.
* Rows can be label with more than one name, [name1 name2] that are before or after the size of the row;
* Grid-column shorthand: grid-column: 1 / -1;
* Grid-area shorthand: grid-row-start / grid-column-start / grid-row-end / grid-column-end;
* Gaps can be applied with: grid-gap: grid-row-gap grid-column-gap;
* To position elements, a grid-area can be defined inside each item, but other (the best one of all) approach could be: define areas in the container, with grid-template-areas (all the cells need to be named, or filled with a dot).

Names can be assigned an repeated when generating rows:  grid-template-columns: repeat(4, [col-start] 1fr [col-end]);

These automatic generated lines are consumed: grid-column: col-start 1 / col-end 2;

* The number after the line names are occurrences (the first, the second, and so on).
* Areas can be defined in the row and column creation statements (e.g element2 area) :

grid-template-columns: [element2-start] repeat(4, [col-start] 1fr [col-end])  [element2-end] ;

And consumed like this by the element: grid-area: element2

* Only static elements are part of the grid. E.g a header with position of fixed is not included.
* When defining rows in the grid, Fit-content(8rem) is like an auto, but it ensures a min-height. Only one “auto” is allowed when defining rows.

To change the behavior of the all elements inside each cell,

For the x axis justify-items: center;

Stretch is the default

For the y axis align-items: center;

Stretch is the default

To change the behavior of an element:

justify-self: center;

align-self: center;

To move the grid inside the container, (usually, when the container is bigger than the grid) apply: justify-content: center; or start end, and so on

* The best approach: define areas in the container, and consume them in the elements, and use media queries to change the areas.

When automatic rows are created in the grid, a way to control them is:     grid-auto-rows: minmax(20rem, auto);

* By default, new items are added to a new row, but can be added to new columns (not recommended) using: grid-auto-flow: columns.

To have the responsive action like flexbox, to generate columns dynamically, depending on the viewport width, use: grid-template-columns: repeat(auto-fill, 10rem);

* If there aren’t enough items to fill all the columns created, it’s better to use auto-fit instead, which centers the content instead of creating more columns

If an element wasn’t assigned a specific area, and there’s a space available in an upper cell, the elements below in the DOM take that space; if the element was assigned an area, that space it’s left empty. But can be filled if: grid-auto-flow: row dense;

}

* CSS vs Flexbox: 2 directions, use the grid; for one direction (like placing elements next to each other, or below) use flexbox;
* Instead of using margin inside the elements, it’s better to add gaps between the cells.

Transform

* Use transform to just visually move elements, but keep them in the document flow.
* When rotating the element, axis X and Y are rotated as well
* Images can be zoomed by using scale(1.5);
* With skew a rombo shape can be achieved.
* To rotate in 2D, apply rotateZ, like a piece of paper pinned on a board, and for 3D rotations, rotateX and rotateY.
* Transform: perspective(100px) it’s like seen the element 100px away from it. Other way is to apply the perspective attribute to the container, perspective: 100px, and is the same for all child (recommended approach). Perspective origin can be used to adjust the perpective as well, movin around the element (right, left, top, etc).

Flattens of the 3D space by the parent: it’s the default behavior. E.g when you turn the container 90 deg in the Y axis, nothing can be seeing inside it, even when you have self rotated elements inside (that should emerge). To avoid this default, apply this to the container: transform-style: preserve-3d;

* To not show the back of an element, apply backface-visibility: hidden;

Transitions

* Transition between opacity: 0 to opacity:1. Not all properties can be watched and be trasitioned. All properties can be watched, or max 4 named properties.
* transition: WHAT DURATION DELAY TIMING-FUNCTION; transition: opacity 200ms 1s ease-out;
* ease-out starts fast and end slow.
* cubic-bezier function can be change in Chrome dev tolos, to achieve a bouncing effect.
* Bug in the code: the modal is always on the document flow, but with opacity 0, so you can’t click the Choose plan button at the position of the modal.
* The display property can’t be transitioned.

Animations

Better and less syntax than transition. Animate between 2 states. Better control over the transition. Example:

.main-nav\_\_item--cta {

    animation: wiggle 200ms 1s 8 alternate;

}

@keyframes wiggle {

    from {

        transform: rotateZ(0deg);

    }

    to {

        transform: rotateZ(10deg);

    }

}

* By default, the animation final state is not kept by the element. (the cta button stands in the horizontal position after wiggling). Removing the reverse value, and add an animation-fill-mode; adding “forward” makes the element to keep the final state, or “backwards”, the starting position before the animation is gonna be the initial state. Applying “both” will combine the 2 behaviors above.
* The transition between each state can be controlled with timming functions, like ease-out, for example.

If the element has a transform property in the default state, like skew(20deg) and a transform property is applied in the frames, that transform property should be added in the frame, to not loose it.

.testimonial\_\_image-container {

    width: 100%;

    max-width: 40rem;

    box-shadow: 3px 3px 5px 3px rgba(0,0,0,0.3);

    transform: skew(20deg);

    overflow: hidden;

  }

  .testimonial\_\_image-container:hover {

    animation: flip 2s  forwards 1;

  }

  @keyframes flip {

    from {

      transform: rotateY(0deg) skew(20deg);;

    }

    to {

      transform: rotateY(360deg) skew(20deg);;

    }

  }

The animation phases are events that can be listened to and trigger other actions: ctaButton.addEventListener('animationstart', (event) => {

    console.log('Animation started', event);

});

ctaButton.addEventListener('animationend', (event) => {

    console.log('Animation ended', event);

});

ctaButton.addEventListener('animationiteration', (event) => {

    console.log('Animation iteration', event);

});

* Root: { --dark-green: #someGreenColor} variables are defines at the root document level and then consumed in the selector like var(--dark-green); IE browser doesn’t support it, but Saas can be used instead;

Tips

* To support older browsers, use vendor prefixes, like display: -webkit-box;
* Use autoprefixer online || with webpack; to add the needed prefixes and avoid checking can I use.
* To avoid implementing a feature like css, and all it’s associated css rules applied, like width of 100% to elements, or fonts, it’s better to check if the browser supports the grid feature.
* Polyfills are js packages that run in the browser and can mimic some not supported features with other css features (in hacky ways). The downside is that the have to run in the browser, impacting performance.

To reset have consistent style across browser, applying [npm](https://www.npmjs.org/package/normalize.css/) install normalize.css, but this changes default styles that you may not even use in the app.

* To name classes:
* 1) use kebab-case because css is case insensitive;
* 2)don’t add a property to the name, like .title-blue, instead .page-title
* Follow block level modifier BEM styles:
* 1).BLOCK\_\_ELEMENT—MODIFIER , .main-menu\_\_item—highligted, or like .plan—highlighted, in the project.

Sass

* It doesn’t run on the browser, just use in development, and needs to be compiled to css before production
* Use sass –watch main.scss main.css to automatically update the css file upon changes.
* Lists can be stored in variables, like 1px solid blue, or Montserrat, sans-serif
* It’s better to use map instead of single variables (like colors), so it’s like having an array, and each variable can be accessed individually.
* The lighten function is super useful, because it lightens a color that can be changed later on.
* The best is to complile one scss file into a one css file, which is then requested by the browser. Use partials (\_fileName, which are just used in the compiled step ) and better imports of scss files.
* Media queries inside the components, mind blowing!
* With inheritance you can create a rule and them apply to other elements, without the need of adding it to the html elements, it’s like an internal class.
* No need to add new rules in a separate line for :hover and :active, just nest them inside the element with &:hover and &:active;