Net Ninja CSS Flexbox and Grid Notes

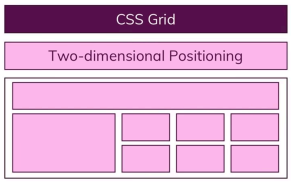
Last Updated: December 25, 2020

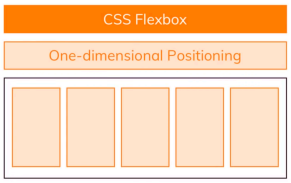
Sources

- [YouTube] Academind: CSS Grid vs Flexbox (Link)
- [YouTube] Net Ninja: CSS Flexbox Tutorial (Link)
- [YouTube] Net Ninja: CSS Grid Tutorial (Link)
- [YouTube] Net Ninja: Mobile-First Responsive Build (Link)

CSS Grid vs Flexbox

- CSS Grid is a relatively new feature of the CSS specification that allows you to position elements as a 2D grid on the page.
    - This flexibility allows us to show the page in the way that was intend for it to look.
- Flexbox, on the other hand, is for laying out a single row or column of elements.
    - We can get Flexbox to wrap the elements around to the next line, but this is still 1D behaviour.





CSS Grid

- If, for instance, we have elements such as header, sidebar, main, and footer nested within a <body> tag, we can use the CSS Grid functionality to position these elements.
    - The display: grid property can be set within the body to turn the body into a grid.
    - Setting the grid-template-columns and grid-template-rows properties defines the amount of space that the columns and rows should take up on the page.
- Note that rows do not have to be explicitly defined, since CSS Grid can automatically create the rows.
    - However, it can be useful to explicitly define the row areas.
- After defining the space that the columns and grid will take up on the page, we can then name these areas by using the grid-template-areas property of CSS Grid.
    - This property allows each column and row to be explicitly named and referenced elsewhere in the stylesheet with the grid-area: <area> property.

```css
* {
    box-sizing: border-box;
}

body {
    font-family: sans-serif;
    margin: 0;
    display: grid;
    grid-template-columns: 30% auto;
    grid-template-rows: 60px auto 60px;
    grid-template-areas: "hd hd"
                         "sidebar main"
                         "footer footer";
}

header,
aside,
main,
footer {
    padding: 16px;
    text-align: center;
}

header {
    background: #521751;
    grid-area: hd;
}

aside {
    background: #fa923f;
    grid-area: sidebar;
}

main {
    grid-area: main;
}

footer {
    background: black;
    grid-area: footer;
}

.navigation-list {
    list-style: none;
    margin: 0;
    padding: 0;
    display: flex;
    flex-direction: row;
    justify-content: center;
    align-items: center;
    height: 100%;
}
```

Flexbox

- Flexbox can be used in combination with CSS Grid to great effect.
    - For instance, we can have the elements within a header or footer evenly distributed.
    - This is a perfect case for Flexbox, where CSS Grid would be overkill.
- To use Flexbox, we simply have to add the display: flex property to a container element such as a navigation list for a header.
    - The flex-direction property allows us to define whether the Flexbox is applied as a row or column.
- A common misconception is that we should use either Flexbox or CSS Grid, but the reality is that we should be using both.

CSS Flexbox Tutorial #1 - Introduction

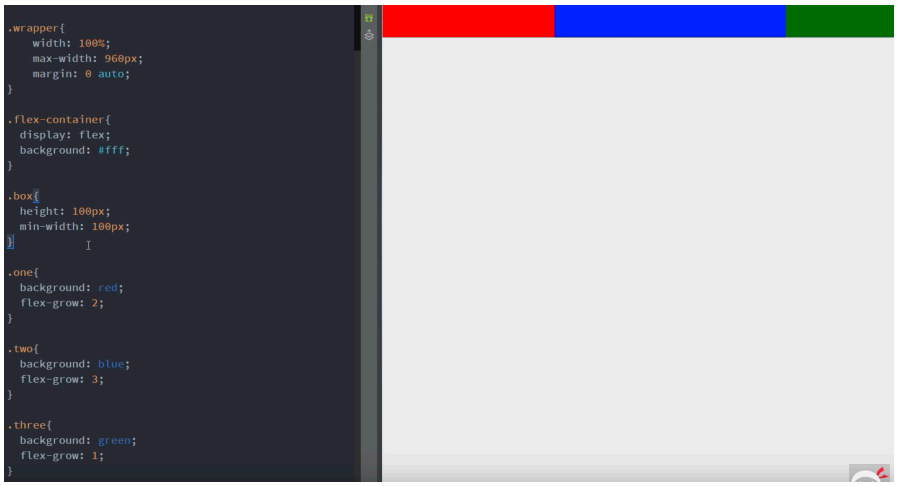
- Prior to Flexbox, we would use properties such as absolute, float, and fixed heights for columns to position elements on the page.
- Flexbox is a CSS display type designed to help us craft CSS layouts in an easier manner.
    - It allows us to control the position, size, and spacing of elements relative to their parent elements and to each other.
    - Flexbox also allows for responsive design.
- For Flexbox to work at the highest level, a property with a display type of flex would have to be added to the parent container.
    - The children elements within the parent element would then be called flex items.
    - We could then control how the elements shrink and grow, as well as the space between them.

CSS Flexbox Tutorial #2 – Flex Containers

- To give the parent container (or wrapper) the Flexbox functionality, we would simply set a CSS property of display: flex for the container element.
    - Now every descendant of that element becomes a flex item.

CSS Flexbox Tutorial #3 – Flex Grow

- Since the child items are not necessarily sized to the full container height or width, we may want to permit them to grow.
    - The flex-grow property allows these items to grow to take up the available space in the container at whatever rate is specified in the property.
- This flex grow behaviour is quite similar to what we see with the Bootstrap grid system, where we have twelve total columns to which elements can be sized to.

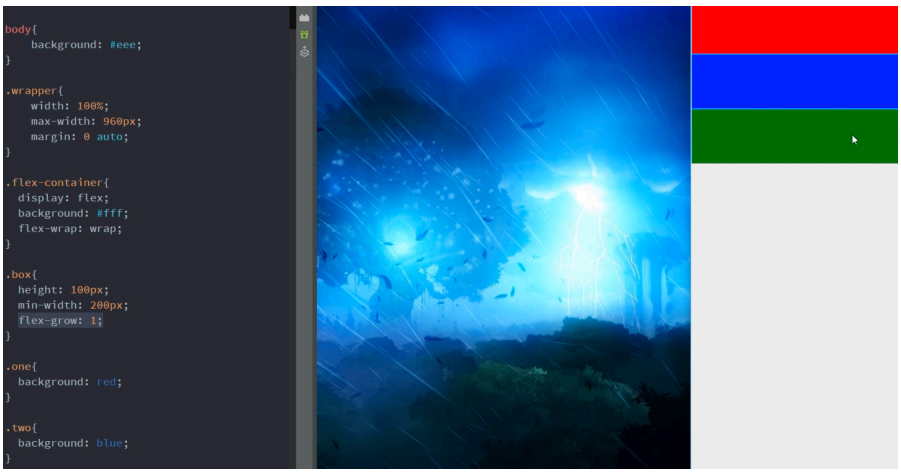


CSS Flexbox Tutorial #4 – Flex Shrink

- Flex shrink is just the opposite of grow, where the rate at which elements shrink is instead provided to a flex-shrink property.
    - This is useful if we have set the initial width of each of the element, but then need the elements to shrink at different rates as the width of the container window (or browser) decreases.
- This shrink functionality is rarely used in practice.

CSS Flexbox Tutorial #5 – Flex Wrap

- In a situation where a minimum width has been set for elements that are adjacent to each other and the browser width is decreased below that aggregate width, the elements will start being shifted off-screen.
    - This is not ideal, so the flex-wrap property can be used to push the right-most element



(or left-most, if wrap-reverse is set) to the next row where it would take up the available space of the container.

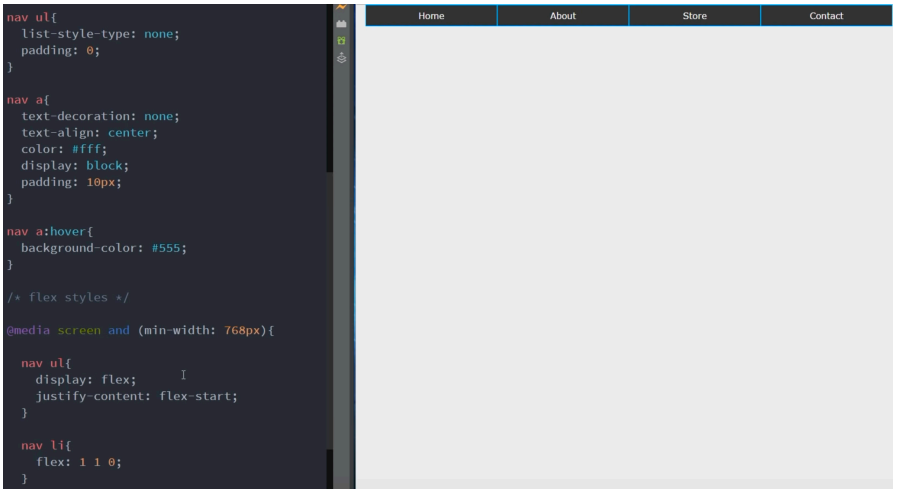
- This is a great way to add responsiveness to a design without having to resort to media queries.

CSS Flexbox Tutorial #6 – Flex Basis

- The flex-basis property is similar to using a minimum width property on elements, where the elements all start at a width of whatever value is provided to the property.
    - The difference between flex-basis and min-width is that the minimum width property will cause elements to be pushed out of the current viewport, requiring scrolling in the browser, while the flex basis property will shrink the elements once the width value is hit so that they do not get pushed off-screen.
- Flex basis should generally be used in place of a minimum width property since it will increase the responsiveness of the page.
- Flex-grow, flex-shrink, and flex-basis can all be combined into one shorthand notation, simply called flex.
    - The syntax for the combined property is flex: <grow> <shrink> <basis> and is the best-practice way to use these properties.
- If we see something like flex: 1, it means that the growth is '1', the shrink is '0', and the flex-basis is set to '0px'.

CSS Flexbox Tutorial #7 – Creating a Menu with Flexbox

- The @media property can be used to apply a specific set of CSS properties only when the screen width has decreased below a specified value.
- A property called justify-content can be used along with its flex-based values to flex content to the ends of the container.
    - This can be especially useful for navigation bars or footers.
- Using flex, we can avoid the problems that generally pop up when dealing with float and can do so with a minimal amount of code.



CSS Flexbox Tutorial #8 – Creating Nested Menus with Flexbox

- Keep in mind that we can use the Flexbox in a nested manner, where a container can use display: flex and then the children can also apply that same property.
    - This allows us to have nested flex elements and items.

CSS Flexbox Tutorial #9 – Flow & Axis

- The flex-flow: column property will allow us to stack flex items in a column rather than a row.
    - When we set the flex flow property to column, we are changing the main axis of the flow from horizontal to vertical.
    - Elements always follow the flow of the main axis, so they now flow downwards in a column.
- There are certain properties when we use Flexbox that only apply to the main axis, and other properties that only apply to the cross axis.
    - For instance, the justify-content property only applies along the main axis.

CSS Flexbox Tutorial #10 – Align Items on the Cross Axis

- If we use the align-items property, we can alter the position of elements along the cross axis instead of the main axis.
    - Recall that the justify-content property alters the position of elements along the main axis.
    - Keep in mind that when we set the flex-flow: column property, the directions that the align-items and justify-content properties position the elements in will be rotated.

CSS Flexbox Tutorial #12 – Element Order

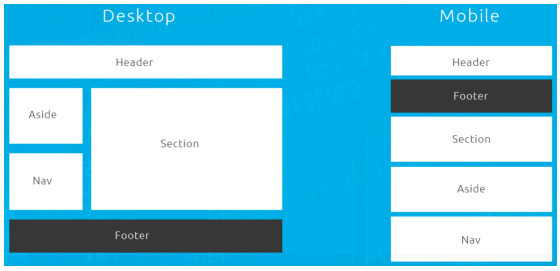
- Flexbox provides an order property that allows us to change the order of elements within the Flexbox container.
    - The elements that assigned the largest values for the order are positioned to the right, no matter how they are ordered in the html.
- Note that order values can be less than 0 as well, and it's not uncommon to see a value of -1000 for items that need to be shown on the left.
    - It is not necessarily best practice to set the order in this manner, though.

```
body{
  background: #eee;
  color: #333;
}

.wrapper{
  width: 100%;
  max-width: 960px;
}

#blocks{
  display: flex;
  margin: 10px;
  justify-content: space-between;
}

#blocks div{
  flex: 0 0 100px;
  padding: 40px 0;
  text-align: center;
  background: #ccc;
}

.one{order: 3;}
.two{order: 2;}
.three{order: 1;}
.four{order: 0;}
```


CSS Grid Tutorial #1 – Why Use CSS Grid?

- While float used to be the way to position elements and sections on the page, that is now an archaic way of positioning.
    - Both Flexbox and CSS Grid have replaced the functionality that was previously obtained from float.
- Flexbox was the first to take over from float, but it had its own problems since it only dictated positioning in one dimension.
    - In order to create separate sections, wrapper div elements would be needed to nest Flexbox items.
    - Ideally, we would not have the extra markup elements (div tags) that were required to have sectioned off areas with float and Flexbox.
- CSS Grid substantially reduces the complexity of CSS markup by removing the need for the vast majority of nested elements and wrapping divs.
    - As well, the order of the elements in the HTML does not matter when CSS grid is being used.
    - Content can be entirely rearranged depending on the screen size.
- The webpage is broken down by CSS grid into a grid of rows and columns that make up areas.

```
    footer{
      grid-column: span 3;
      background: #a56dda;
    }
  </style>
</head>
<body>

  <div id="content">

    <header>Header</header>

    <main>Main</main>

    <aside>Aside</aside>

    <nav>Nav</nav>

    <footer>Footer</footer>

  </div>

</body>
</html>
```



CSS Grid Tutorial #2 – Columns

- We can have as many columns as we would like in our CSS Grid.
    - The first step is setting the number of columns and the width of each column.
    - Then we can tell elements how many columns that they can span and specify which columns.
- Columns can be equal widths, or they can be set to different widths.

- To set the wrapper to display a grid, we simply add a property of display: grid into the #content tag (or whatever we call our CSS Grid wrapper).
    - The grid-template-columns property will then be added to that same tag to specify the number of columns that we want to have, along with their relative widths.
- Child divs are automatically placed into the columns in the order that they are entered, provided that no further grid area assignments have been provided to the children.
- Providing fractions to the grid-template-columns property in the form of 1fr, for example, can be done in place of percentages for each column.
    - The repeat(<times>, <fraction>) function can also be used to provide a more succinct grid column template, which is probably a more best practice way of defining columns.

```
<!DOCTYPE html>
<html>
<head>
	<title>Using CSS Grid</title>
	<style>
		body{
			color: #fff;
			font-family: 'Nunito Semibold';
			text-align: center;
		}
		#content{
			display: grid;
			/*grid-template-columns: 30% 20% 50%;*/
			/*grid-template-columns: 1fr 2fr 1fr;*/
			grid-template-columns: repeat(3, 1fr);
			max-width: 960px;
			margin: 0 auto;
		}
		#content div{
			background: #3bbced;
			padding: 30px;
		}
		#content div:nth-child(even){
			background: #777;
			padding: 30px;
		}
	</style>
</head>
<body>

	<div id="content">

		<div>1</div>
		<div>2</div>
		<div>3</div>
		<div>4</div>
		<div>5</div>
		<div>6</div>
		<div>7</div>
		<div>8</div>
		<div>9</div>

	</div>

</body>
</html>
```

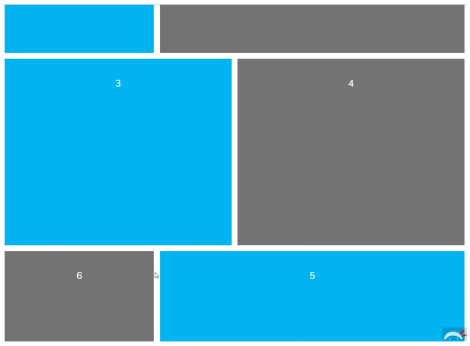
CSS Grid Tutorial #3 – Rows

- If elements are not explicitly placed into areas, they will be wrapped beyond the specified number of columns into new rows.
- Row heights will automatically adjust to fit the content by default, but we can override this behaviour by providing a grid-auto-rows property inside the wrapper declaration block.
    - Keep in mind that this will disregard the height of any content inside the grid cell.
    - Assigning a minmax(<min><max>) function to the property will set a minimum height for each row along with allowing the row to expand to fit whatever content is provided.
- A grid-template-rows property is also available that allows us to set the number of rows along with the height of each row.
    - This property could be assigned the repeat(<times>, <fraction>) function, just as we did before for the template columns.
    - Note that the minmax(<min><max>) function can be provided inside of the repeat(…) function.
    - This property is not often used since we typically just want rows to be created automatically as we add more content.
- The column-gap, row-gap, and gap properties can be set inside of the wrapper declaration block.
    - These properties will provide gaps between the rows and columns of the grid without applying to the outside of each grid cell, which would happen if we used the padding property instead.

```
body{
	color: #fff;
	font-family: 'Nunito Semibold';
	text-align: center;
}
#content{
	display: grid;
	grid-template-columns: repeat(3, 1fr);
	/*grid-auto-rows: minmax(150px, auto);*/
	grid-template-rows: repeat(3, minmax(150px, auto));
	/*grid-row-gap: 10px;
	grid-column-gap: 10px;*/
	grid-gap: 10px;
	max-width: 960px;
	margin: 0 auto;
}
#content div{
	background: #3bbced;
	padding: 30px;
}
#content div:nth-child(even){
	background: #777;
	padding: 30px;
}
```

CSS Grid Tutorial #4 – Grid Lines

- Grid lines will help us to define positions on a grid and will help us to place elements into whatever sections of the grid that we want.
    - These lines are shown in both the column and row directions.
    - As a general rule of thumb, if we have ‘x’ columns, we will have ‘x+1’ column lines.
- These column and row lines can now be used to dictate where elements will be placed inside of the grid.
    - For instance, we will specify from which row lines the element should start and end, along with which column lines.

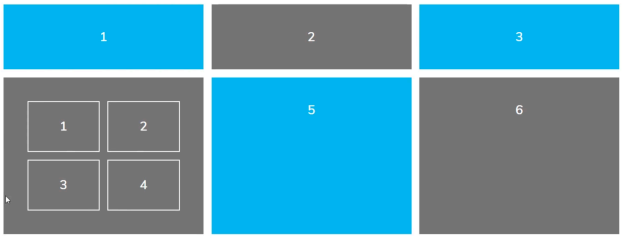



```
.one{
	/*grid-column-start: 1;
	grid-column-end: 3;*/
	grid-column: 1 / 3;
}
.two{
	grid-column: 3 / 7;
}
.three{
	grid-column: 1 / 4;
	grid-row: 2 / 4;
}
.four{
	grid-column: 4 / 7;
	grid-row: 2 / 4;
}
.five{
	grid-column: 3 / 7;
}
.six{
	grid-column: 1 / 3;
	grid-row: 4;
}
```

- Each element inside of the grid wrapper tags (i.e. #content, in our example) can have grid-column-start and grid-column-end properties specified inside of its CSS selector declaration block.
    - A shorthand for specifying both the start and the end is grid-column: 1 / 3, which means that the element will stretch from grid line '1' to line '3' (note that '1' is the first line on the left).
    - This same concept is also applied to grid rows.

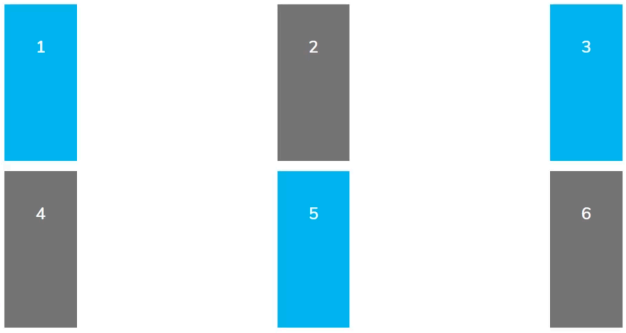
CSS Grid Tutorial #5 – Nested Grids

- We can nest a CSS Grid inside of an element that is already a grid element.
    - A property such as grid-column: span 3 can be used in place of explicit grid lines to stretch an element across multiple grid cells.



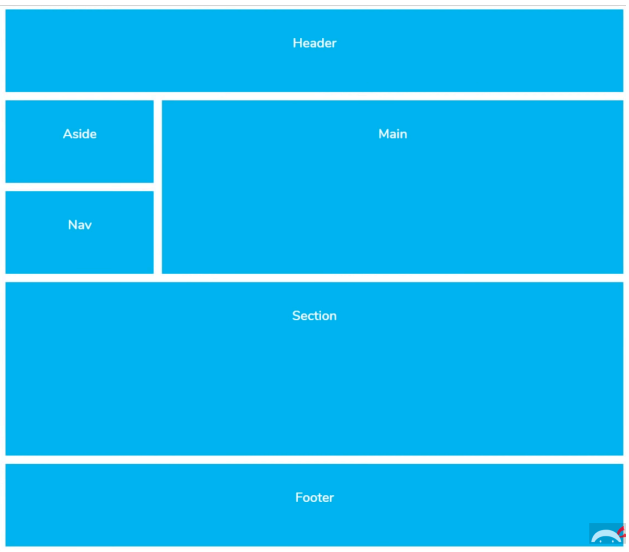
CSS Grid Tutorial #6 – Aligning & Justifying Items

- Including the align-items property inside of the wrapping selector's declaration block can allow us to align the content within each grid item to the top, bottom, etc., rather than filling the entire grid cell.
    - Note that align-items: stretch is applied by default to the elements within a grid cell.
    - This property allows us to position the items vertically within a grid cell.
- To position items horizontally within a grid cell, we can use justify-items instead.
    - For example, justify-items: start will position items to the left side of the grid cell.
- Each element can have these properties applied to it individually inside of a grid cell as well.
    - In this case, the properties would be align-self and justify-self instead.
    - Using align-self: center and justify-self: center can be quite powerful for centering elements within the grid cell.



CSS Grid Tutorial #7 – Create a 12-Column Grid

- We can use CSS Grid to create a 12-column grid similar to that used by the Bootstrap Grid System.
- While we should be setting the number of columns using grid-template-columns, we don't need to set the number of rows in the same way since these can be automatically defined or defined by us when we use the grid-row property inside of each grid item's declaration block.
- Note that the major browsers now support a grid line overlay so that we can see where we are positioning elements within the CSS grid.




```
header{
    grid-column: 1 / 13;
}
main{
    grid-column: 4 / 13;
    grid-row: 2 / 4;
}
aside{
    grid-column: 1 / 4;
}
nav{
    grid-column: 1 / 4;
}
section{
    grid-column: 1 / 13;
    grid-row: 4 / 6;
}
footer{
    grid-column: 1 / 13;
}

#grid{
    position: absolute;
    top: 0;
    left: 0;
    display: grid;
    grid-template-columns: repeat(12, 1fr);
    grid-auto-rows: minmax(100%, auto);
    width: 100%;
    height: 100%;
    background: transparent;
    padding: 0;
    display: none;
}
```

CSS Grid Tutorial #9 – Grid Areas

- Grid template areas are probably the most powerful feature of the CSS Grid layout.

- Each of the elements (e.g. header, main, section, footer, etc.) are provided with a grid area name, which we can then use to position the element inside of the grid.
    - The property that we use is grid-area, and we assign a name as the attribute (e.g. grid-area: header).
    - These grid area names do not have to be the same name as the tag – they can be whatever we decide makes the most sense.
- Now we can define a grid-template-areas property in the wrapping div's declaration block, where we define where each of the grid areas will be positioned in an ASCII-style format.
    - If we do not want to assign a grid area to a portion of the layout, we can place a period (.) in place of an area name.

```
#content{
    display: grid;
    grid-template-columns: repeat(4, 1fr);
    grid-auto-rows: minmax(100px, auto);
    grid-gap: 10px;
    max-width: 960px;
    margin: 0 auto;
    grid-template-areas:
        "header header header header"
        "footer footer footer footer"
        "main main main main"
        "main main main main"
        "aside aside nav nav"
        "section section section section"
        "section section section section";
}
/* desktop grid */
@media screen and (min-width: 760px){
        #content{
                display: grid;
                grid-template-columns: repeat(4, 1fr);
                grid-auto-rows: minmax(100px, auto);
                grid-gap: 10px;
                max-width: 960px;
                margin: 0 auto;
        grid-template-areas:
            "header header header header"
            "aside main main main"
            "nav main main main"
            "section section section section"
            "section section section section"
            "footer footer footer footer";
        }
}
        #content > *{
                background: #3bbced;
                padding: 30px;
        }
header{
    grid-area: header;
}
main{
    grid-area: main;
}
aside{
    grid-area: aside;
}
nav{
    grid-area: nav;
}
section{
    grid-area: section;
}
footer{
    grid-area: footer;
    background: #333 !important;
}
```

CSS Grid Tutorial #10 – Responsive Grid Example

- In order to use CSS Grid to make responsive layouts, we can use media queries along with grid template areas.
    - We should leave the default as the mobile view, and then add a media query with the grid-template-areas for the desktop view.
- Keep in mind that the grid template areas have to be continuous rectangular blocks.
- Setting up this way gives a fully responsive layout that continuously adjusts to the display width of the browser and changes the layout once a minimum number of pixels has been reached.

Mobile-First Responsive Build #1 – Introduction

- A useful Visual Studio Code extension that is recommended is Live Server, which creates a local development server with live reload for static and dynamic pages.
- As well, Emmet should be installed for quick boilerplate code.
    - For example, the doc keyword will create boilerplate HTML site code so that we can start working right away.

Mobile-First Responsive Build #2 – HTML Template

- One useful HTML tag is the <a href="#portfolio"> tag, which allows us to navigate to different IDs without our page, rather than just to external links.

```
<li><a href="#portfolio">Portfolio</a></li>
```

Mobile-First Responsive Build #3 – Mobile-First Approach

- It is recommended that we first build our website for mobile devices first, and then gradually build up to larger screens and desktops.
    - That progression might look like mobiles, tablets, laptops, and then desktops with high-resolution screens.
    - Past experience of designers has determined that it is much easier to start with the mobile version and then expand out to desktop-size screens.
- He uses Adobe XD to design both the mobile and desktop experiences before starting with the actual site implementation.
- Media queries are used to setup breakpoints within the website so that different layouts can be used at different resolutions.
    - Since we are taking a mobile-first approach, the site will by default be setup for mobile screen sizes.
    - Media queries will therefore be used for the progressively larger screen sizes.

```
/* small tablet styles */
@media screen and (min-width: 620px){

}

/* large tablet & laptop styles */
@media screen and (min-width: 960px){

}

/* desktop styles */
@media screen and (min-width: 1200px){

}
```

Mobile-First Responsive Build #4 – Base Styles

- One thing to note about modern CSS is that variable support has been added, even without SCSS being used.

- Setting the selector as :root applies the CSS properties (variables) to the root element in the page, which is the <html lang="en"> tag.

```css
/* reset */
body, p, a, ul, li {
  margin: 0;
  padding: 0;
  text-decoration: none;
}

li {
  list-style-type: none;
}
```

```css
:root {
  --primary: #ffc636;
  --secondary: #0a0b5b;
}

.button {
  background: none;
  border: 2px solid var(--primary);
  color: var(--primary);
  padding: 6px 12px;
  border-radius: 20px;
  text-transform: uppercase;
  box-shadow: 1px 2px 3px rgba(0, 0, 0, 0.6);
  display: inline-block;
}
```

- It can sometimes be useful to reset some of the default browser styles, such as body, p, a, ul, and li since the browser has applied its own margin, padding, and text decorations.
- Including an overflow-x: hidden property in the body selector declaration block ensures that the background image will not force the user to scroll horizontally.

Mobile-First Responsive Build #5 – Fonts

- To register font files for use in the applications, we can use the @font-face selector along with the font-family and src properties within the declaration block.
    - The font-family property can now be added to the body or similar top-level selector declaration block to apply the font throughout the page.
- The font-size property can be set to units of em, which is a scalable unit, unlike if we were to specify the size in pixels.
    - The default font size for the browser is 16px, so 1em simply means that the size of the font is 16px * 1.0em = 16px.
    - Now if we were to set the default font-size for the page to be 20px, then the size of the fonts throughout the page would be updated.

```css
/* fonts */
@font-face {
  font-family: 'Rubik Regular';
  src: url("assets/fonts/Rubik-Regular.ttf") format("truetype");
}

h1,h2,h3,h4 {
  color: #DDDBFF;
  font-weight: normal;
  line-height: 1.4em;
}

p,a,li {
  color: #9893D8;
  line-height: 1.4em;
  font-size: 1em;
}

h1, h3 {
  font-size: 1.2em;
}

h2 {
  font-size: 1.6em;
}

h4 {
  font-size: 1.1em;
}

.leading {
  font-size: 1.1em;
}
```

```css
/* large tablet & laptop styles */
@media screen and (min-width: 960px) {
  body {
    font-size: 18px;
  }
}

/* desktop styles */
@media screen and (min-width: 1200px) {
  body {
    font-size: 20px;
  }
}
```

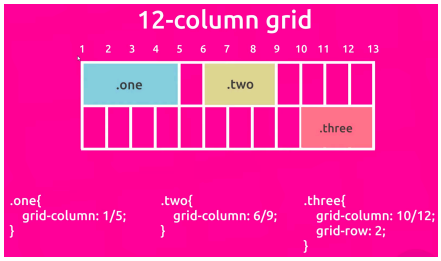
- When it comes to responsive design, it is much easier to work with a base default font size in the body selector, and then use em units for the different elements.
    - Now we can update the default body font size separately in each media query to adjust the text sizes throughout for the different screen sizes.

Mobile-First Responsive Build #6 – Grid Basics

- It is best just to refer to the course notes on CSS Grid above for this section.
- One interesting note is that a stretches across the total width of the page or container it is in because it is a block-level element.
- We can actually see the grid lines when we use inspect functionality in Google Chrome and select the grid container.

Mobile-First Responsive Build #7 – Creating a 12-Column Grid

- Normally, libraries like Bootstrap make use of a 12-column grid.
    - When defining the grid-template-columns in the selector declaration block, recall that we can use the repeat(12, 1fr) function rather than writing out each of the 12 fraction values individually.
- Also recall that we can define where our elements are positioned in the grid by using the grid-column and grid-row properties and defining the grid to and from lines.
    - Note that there is a minor error in the screenshot, where the property for the .three selector should be 10/13.




```css
/* grid testing */
.projects {
  display: grid;
  grid-template-columns: repeat(12, 1fr);
  gap: 10px;
}

.projects a {
  text-align: center;
  background: grey;
  padding: 20px;
}

.projects a:nth-child(1) {
  grid-column: 2/6;
  grid-row: 1;
}

.projects a:nth-child(2) {
  grid-column: 8/12;
  grid-row: 1;
}

You, seconds ago • Uncommitted chan
.projects a:nth-child(3) {
  grid-column: 5/9;
  grid-row: 2;
}
```

- The nth-child(<number>) pseudo-selector can be useful for selecting certain tags within the application for which we can apply a specific style.

Mobile-First Responsive Build #8 – Mobile Styles (Part 1)

- The grid selector can be used in numerous HTML tags, wherever it makes sense to have a grid in place.
- Some developers do not agree with using id selectors at all, but Shawn does not agree.
    - He believes that it is fine, so long as they are used properly.
- The way that he has designed his web app is to first get all of the HTML in place and properly organized before proceeding, section-by-section through the CSS styling.
    - It is fine to go section-by-section for both HTML and CSS as well, as long as the mobile-first responsive design is the priority before moving on to larger screens.

Mobile-First Responsive Build #10 – Mobile Styles (Part 3)

- When sizing elements inside of containers, it is a good idea to use percentage values for the width property since it will allow responsiveness when the screen is resized.
- Both CSS Grid and Flexbox can be used at the same time and can be a powerful combination.

```
footer .social {
  grid-column: 7/9;
  display: flex;
  justify-content: flex-end;
}
```

Mobile-First Responsive Build #11 – Tablet Styles

- To start working outwards from the mobile style, simply enable the Chrome developer tools and start increasing the browser window width – the pixel width of the window will be displayed in the top-right corner.
    - Now use the @media screen and (min-width: 620px) selector to begin overwriting selectors from the default mobile style.
- The grid-column: span 4 property is one that he used to say that each HTML element should span across four columns automatically.

```
.projects a {
  grid-column: span 4;
  display: block;
  margin: 20px 0;
}
```

Mobile-First Responsive Build #12 – Laptop Styles (Part 1)

- Once the browser window becomes larger than a certain number of pixels – 1060px in this case – we should consider capping the width of the grid container for the site and centering the content.
    - If we don't do this, the width of the content will expand out to the full width of the monitor, which can be tricky for the user to read and doesn't look very good.

```
.grid {
  max-width: 1060px;
  margin: 0 auto;
  width: 100%;
}
```