THIRD EDITION

MASTER FLEXBOX & GRID

How to Create Layouts That Work on Any Screen

UTSAV MEENA

CONTENTS

Before You Begin

This book is divided into two sections:

1. Flexbox
2. Grid

The previous edition of this ebook was 172 pages long. But I got feedback that it's too long and sometimes difficult to follow through.

This edition is shorter and skips less important details, like how CSS processes things internally or the complex math behind them.

Most importantly, it's focused on visuals, not text.

But if you still want the previous edition, you can email me at utsav@flexicajourney.com.

Second thing: I assume you already know some basic CSS, like general CSS syntax, common properties, and their meanings.

I know it's cliché, but just reading won't make you great at something (at least not much).

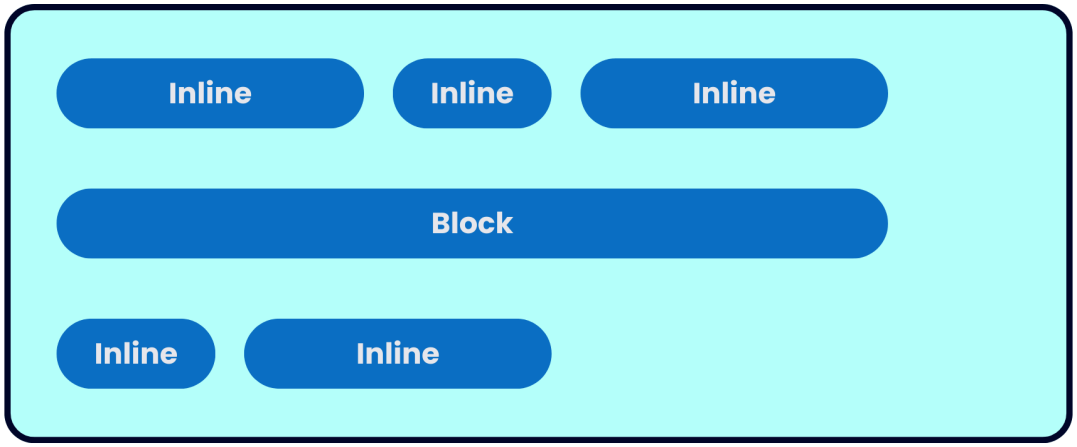
So, you need to practice. I've added CodePen links for all the examples so you can experiment. Plus, whenever you learn a new property, try it out with a simple example.

With that said, let's get into it.

Flexbox

Do you know how *inline* vs. *block* elements are different?

See, this:



Simply put, *inline* elements are good neighbors; they let others stay beside them in the same line.

Block elements are like the rude ones; they take up the entire horizontal space, leaving no room for others.

But how's it related to Flexbox?

You decide between *block* and *inline* for an element by the property ***display***.

This exact property is used to make any element a Flexbox.

This means we neither set it to *inline* nor *block*; we set it to flex.

Then the question arises: Do those elements become *inline* or *block*-level elements?

Answer: both. Here's how:

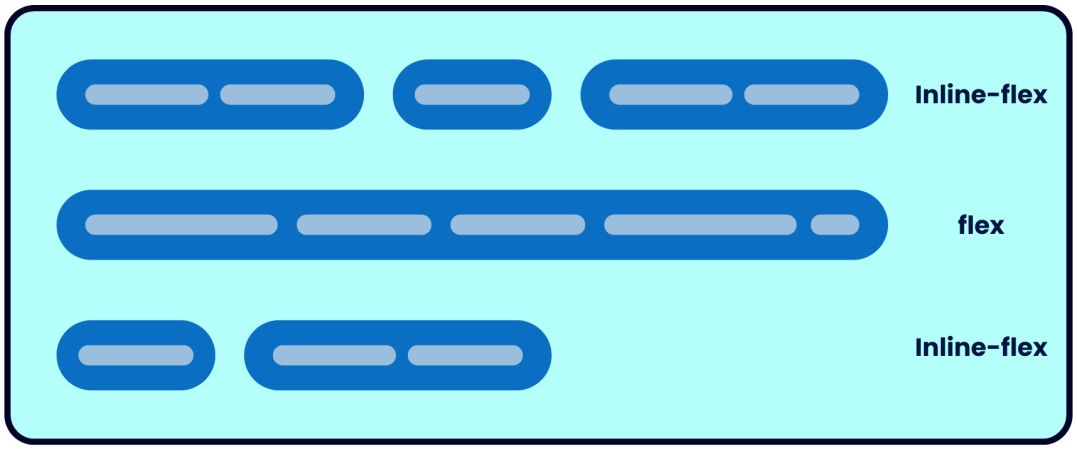
You can declare a Flexbox in two ways:

```
.container{
    display: flex;
}
```

And

```
.container{
    display: inline-flex;
}
```

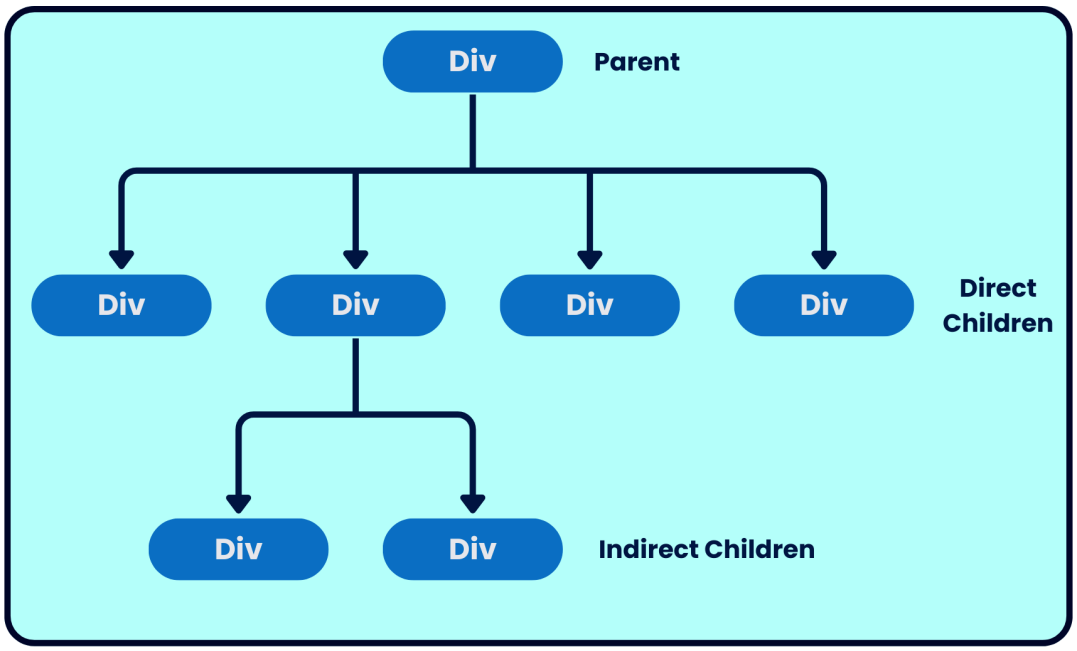
As the names suggest, the first creates a *block*-level Flexbox, while the second creates an *inline* flex. Like this:



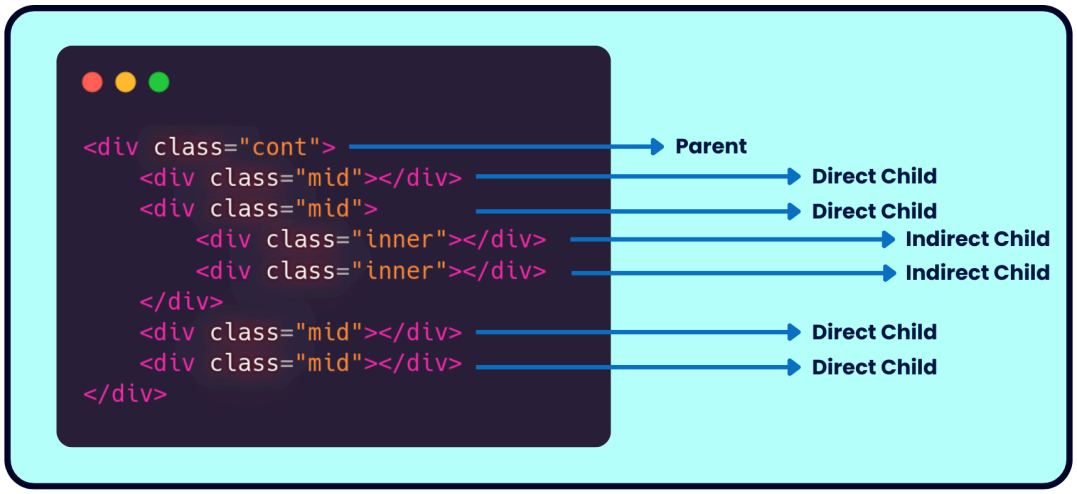
Once you set an HTML element's display to *flex* or *inline-flex*, it becomes a **flex container**.

And all its direct children become **flex items**.

If you don't know what a direct child is, see this:



That’s how it looks in HTML:



It has one exception, though.

If you set any direct child's ***position*** to absolute, it’s no longer a flex item, so the flex properties won’t affect it.

Flex-direction Property

Let’s assume you have this simple navbar HTML:

```
<nav>
    <button>Home</button>
    <button>About Us</button>
    <button>Contact Us</button>
    <button>Privacy Policy</button>
</nav>
```

You make the navbar a flex container like this:

```
nav{
    display: flex;
}
```

From here, the ***<nav>*** is a flex container, and the four ***<button>*** are flex items.

With some basic styling, you’ll have it like this:

Already decent, right?

But what if you want them to align from right to left and not left to right?

Like this:

I have no idea how you'd do that without Flexbox. But with Flexbox, you can use the ***flex-direction*** property like this:

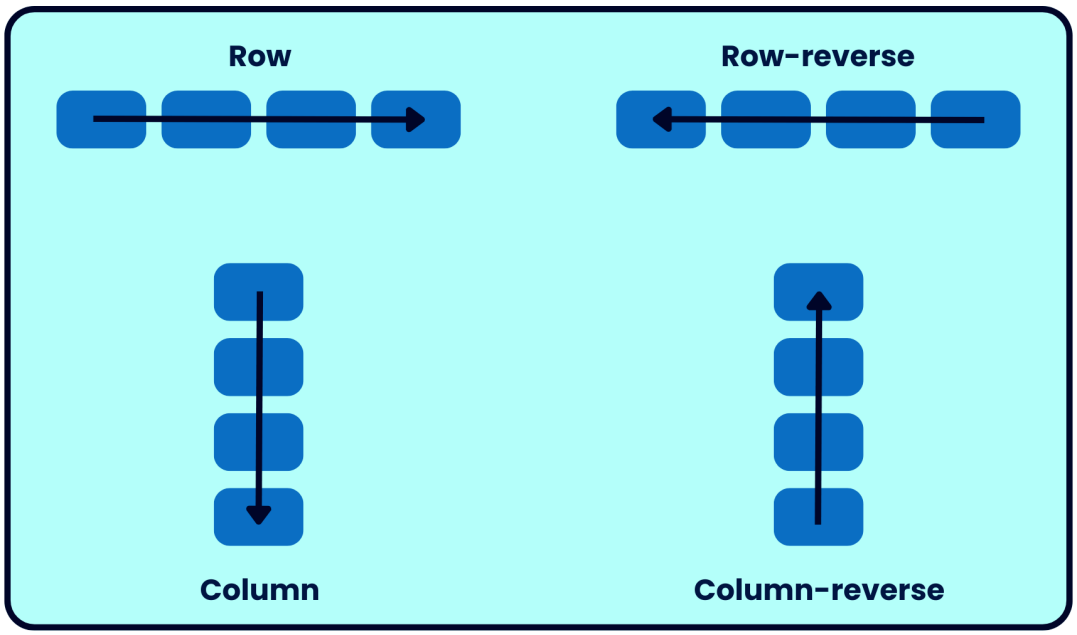
```
nav{

    display: flex;

    flex-direction: row-reverse;

}
```

As the name suggests, ***flex-direction*** lets you decide in which direction flex items should be laid.

It gives you four options:

1. ***Flex-direction: row*** → Left to right (default value)
2. ***Flex-direction: row-reverse*** → Right to left
3. ***Flex-direction: column*** → Top to bottom
4. ***Flex-direction: column-reverse*** → Bottom to top

Like this:



There's a small catch. If your web pages are in English, you don't need to worry about it.

The direction of these four properties depends on the writing mode. Since English is written left to right, *"row"* goes left to right.

But if you change the ***writing-mode*** property to say, *vertical-lr* (which means you're writing vertically), *row* → top to bottom, while *column* → left to right.

And if you want to check out the navbar example live, here's the CodePen link.

Or scan this QR Code:

Gap Property

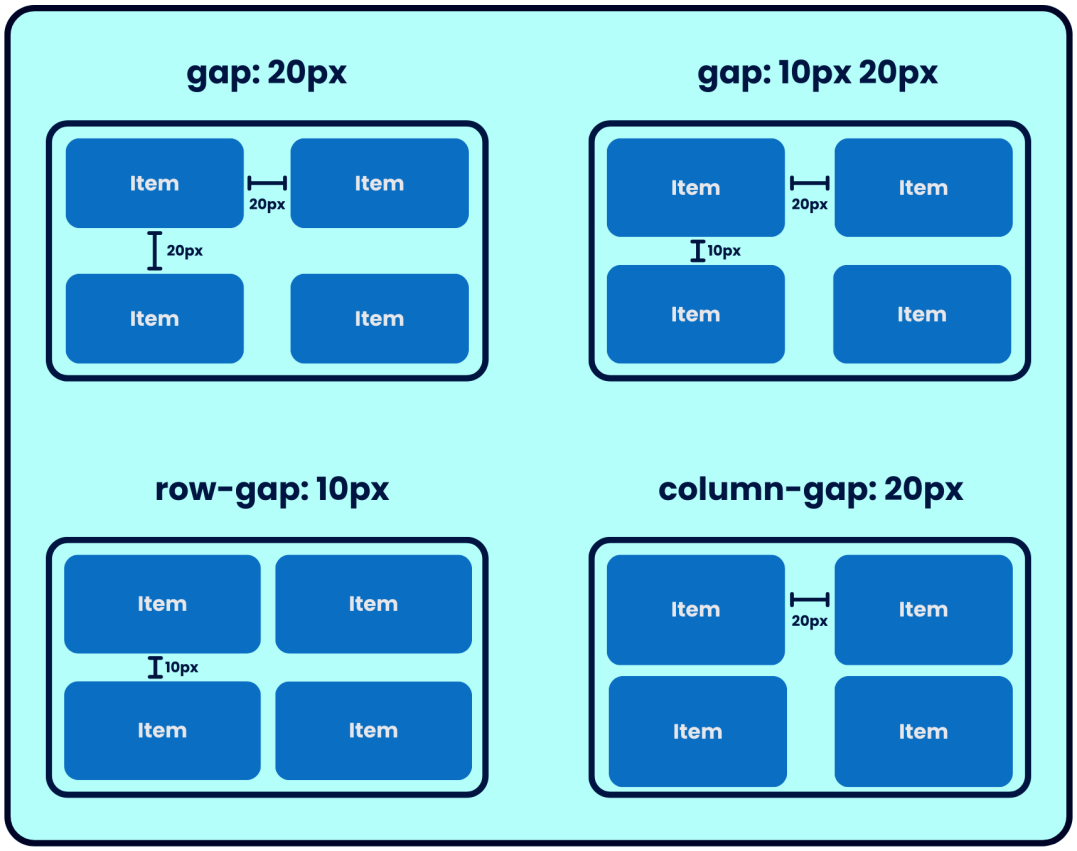
In the last example, we used the property ***gap***. What does it do?

Well, it lets you create gaps between rows and columns.

There are four ways:

1. ***Gap: 10px*** (single value) → Creates a 10px gap between rows and columns.
2. ***Gap: 10px 20px*** (two values) → Creates a 10px gap between rows and a 20px gap between columns.
3. ***Row-gap: 10px*** → Creates a gap between just rows.
4. ***Column-gap: 20px*** → Creates gap between just columns.

Like this:



CSS Tip 💡

In CSS, whenever there's a shorthand for rows and columns, rows come first and columns second. You can remember it with the acronym RC.

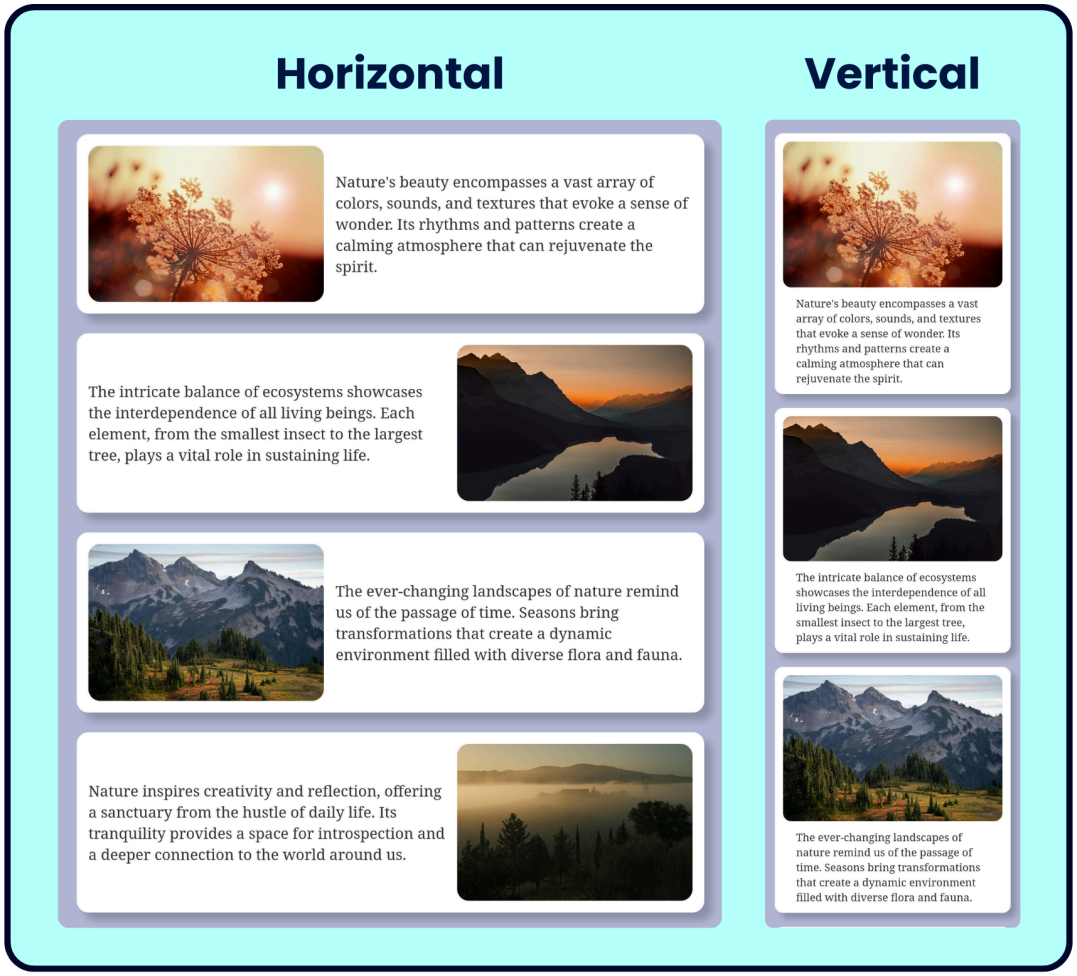
If you want to try this property out yourself, here's a live example on CodePen.

Or scan this QR Code:

Now, let's see how we can make Flexbox a two-dimensional layout.

Flex-wrap Property

Let’s say we need a responsive layout like this:



Each card has an image-text pair, in alternate order, like this:

Image — Text
Text — Image
Image — Text
Text — Image

On vertical (portrait) screens, each of them has an image on top and text at the bottom.

We have this HTML:

```
<div class="cards-cont">
    <div class="card">
        <img src="img1.png">
        <p>Description...</p>
    </div>
    <div class="card">
        <img src="img2.png">
        <p>Description...</p>
    </div>
    <!-- And so on, in same order -->
 </div>
```

Notice that we didn't change the image and text order in the markup because Flex can handle that.

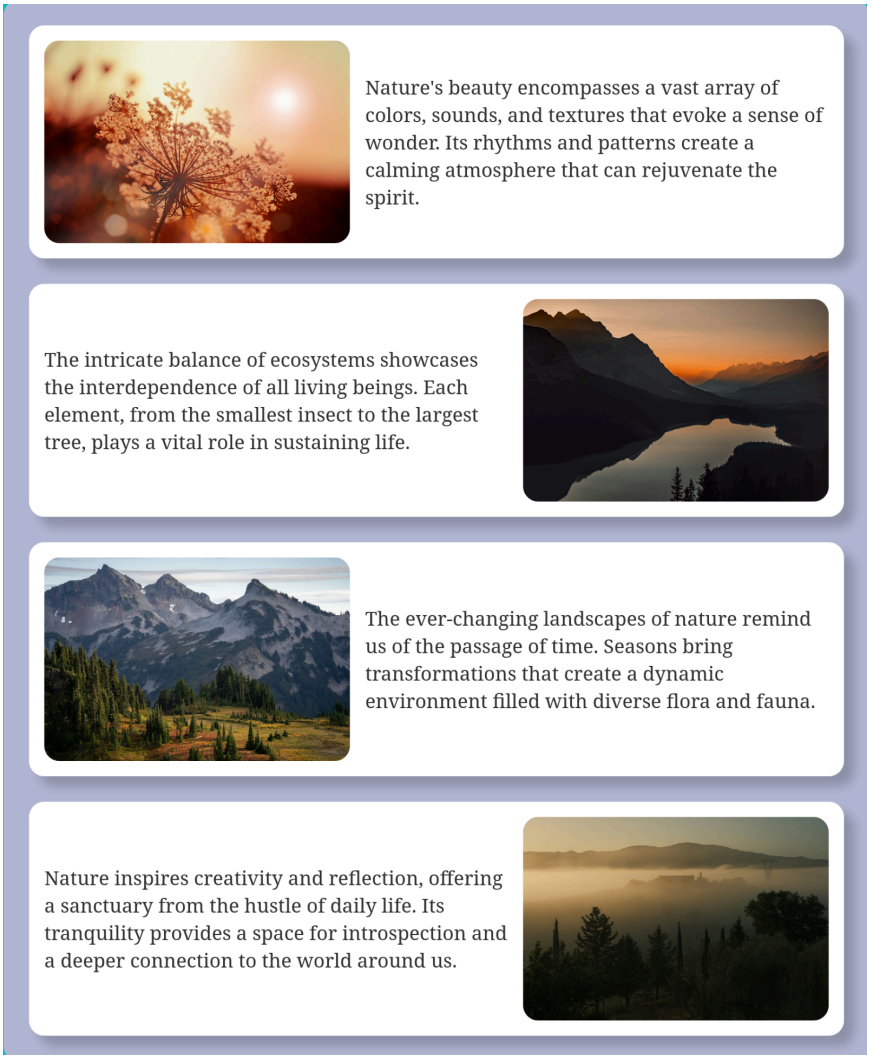
Now, we make the ***.cards-cont*** a Flexbox with ***flex-direction: column***, to lay all four cards vertically.

Then we'll also make the individual cards Flex containers, but for alternate ones, set the ***flex-direction*** to *row-reverse*. Like this:

```
.cards-cont{
    display: flex;
    flex-direction: column;
}
.card{
    display: flex;
}
```

```
.card:nth-child(2n) {

    flex-direction: row-reverse;

}
```

We have the horizontal layout:



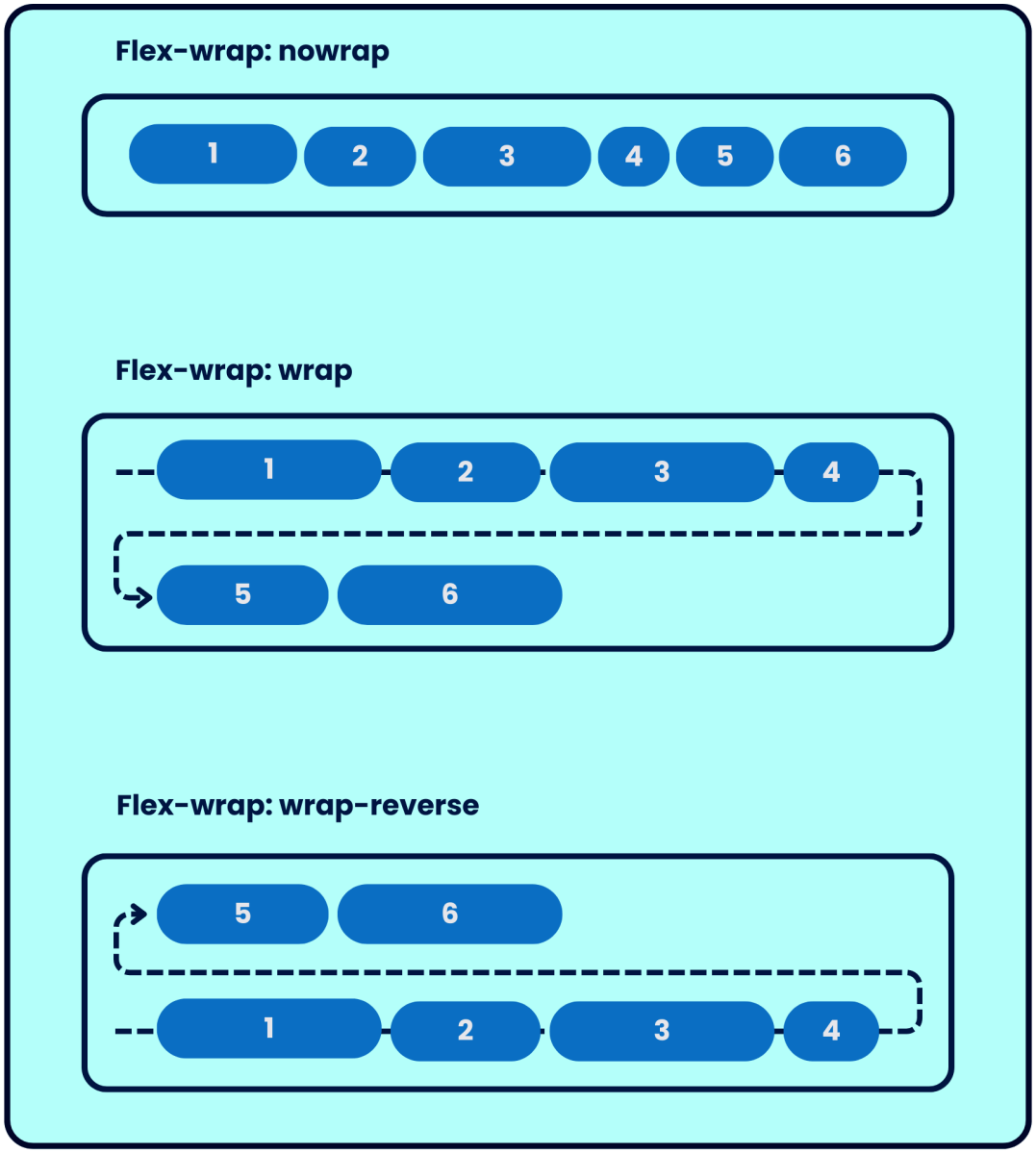
By default, Flexbox puts all the flex items in one *row*/*column* (whichever you set), even if it means shrinking them to fit in.

But you can change this behavior, with the ***flex-wrap*** property.

It can take three values like this:

1. ***Flex-wrap: nowrap*** → Squeezes all the elements in a single row/column (default behavior).
2. ***Flex-wrap: wrap*** → Whenever flex items overflow, put them in new rows/columns next to the previous one.
3. ***Flex-wrap: wrap-reverse*** → When items overflow, put them in a new row/column, but before the previous one.

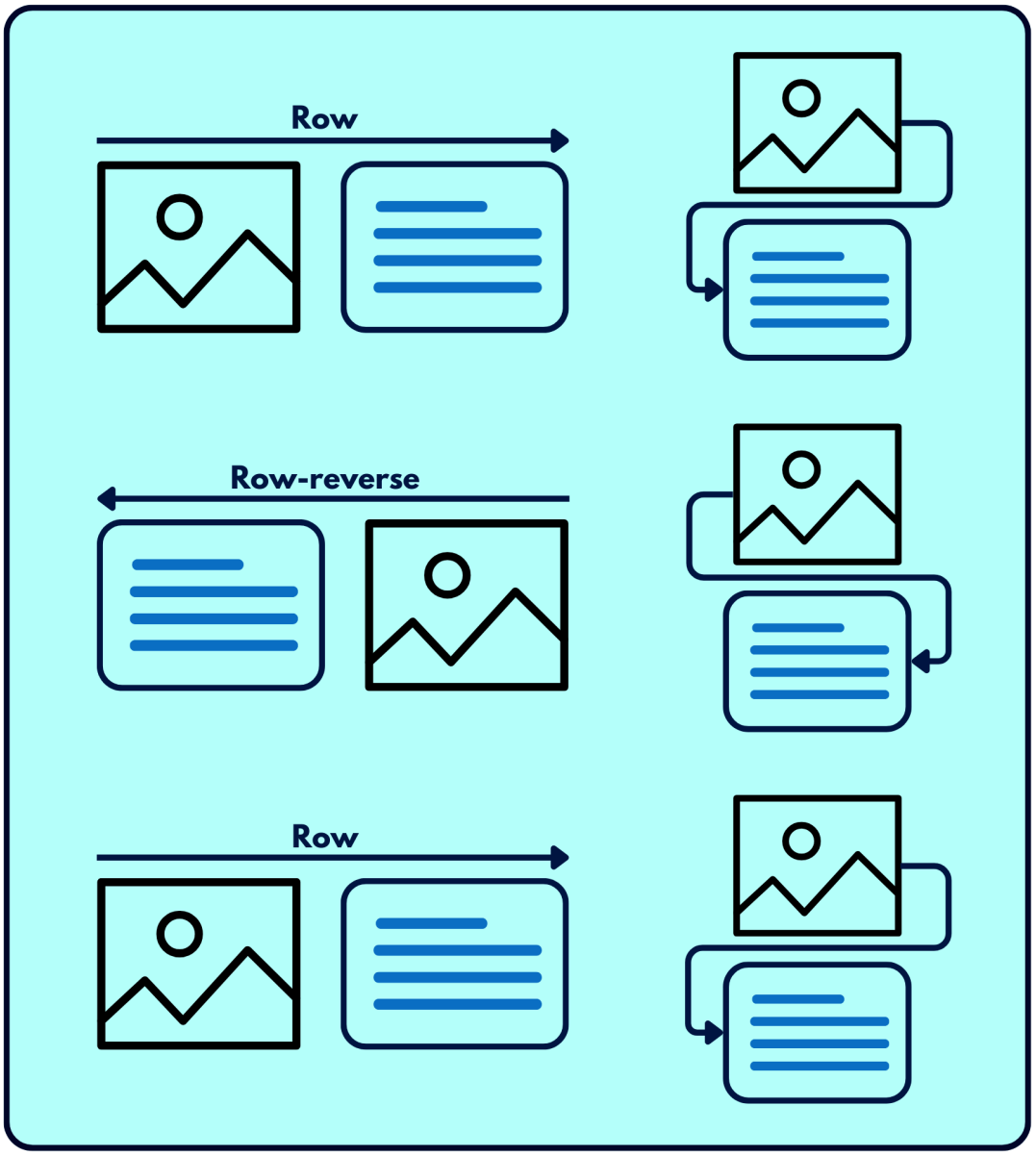
Here's how it looks:



In our layout example, if we also add the ***flex-wrap: wrap***, like this:

```
.card{
    display: flex;
    flex-wrap: wrap;
}
.card:nth-child(2n) {
    flex-direction: row-reverse;
}
```

Then our layout would work this way:

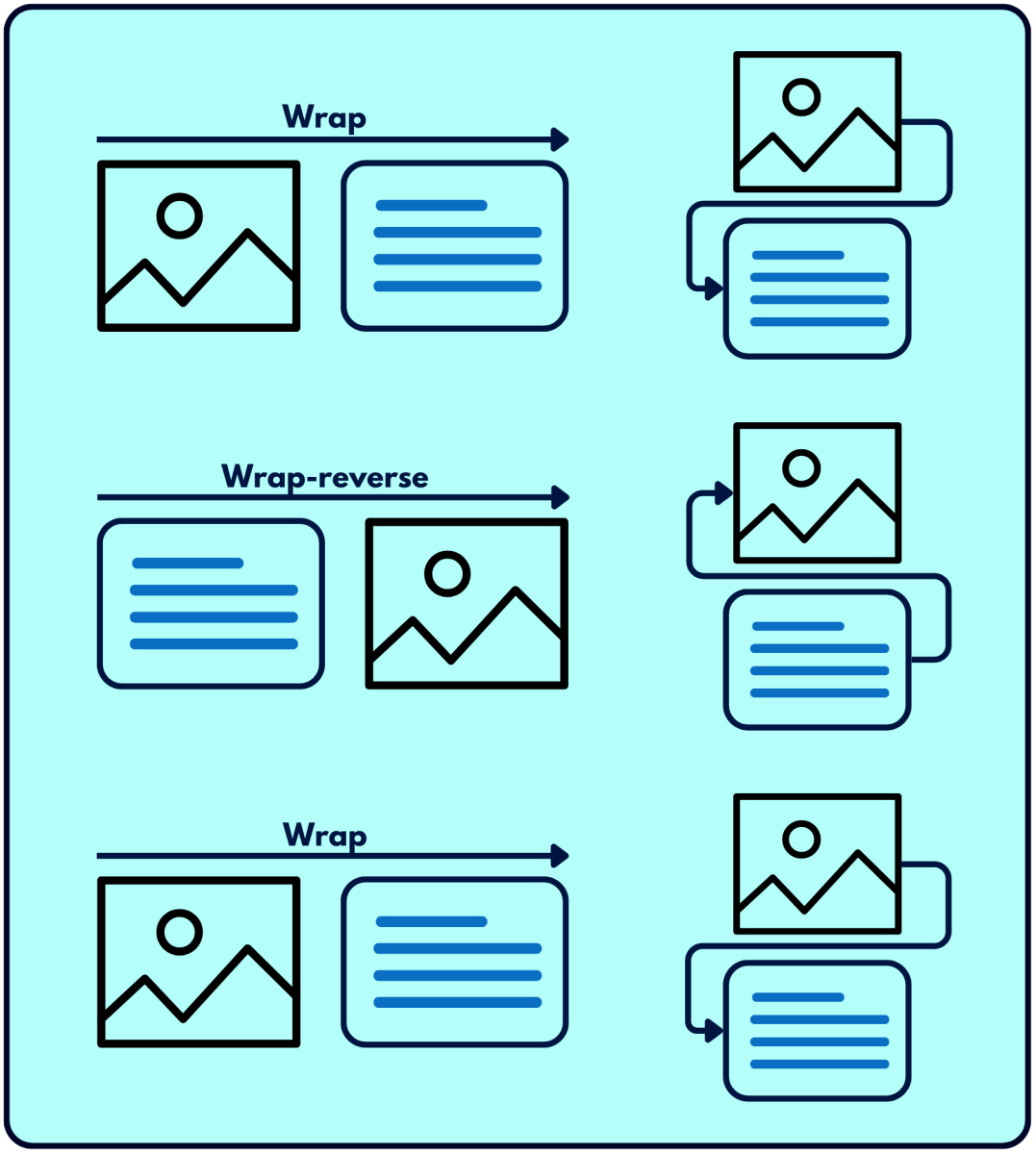


Note 📝

It only wraps when flex items run out of space, so set width or min-width to prevent them from shrinking to zero.

We could also use a different approach.

In the markup, we can alternate the order of the image. Then set the ***flex-wrap*** to *wrap* and *wrap-reverse* alternatively:



Either way, we'll get our desired responsive layout.

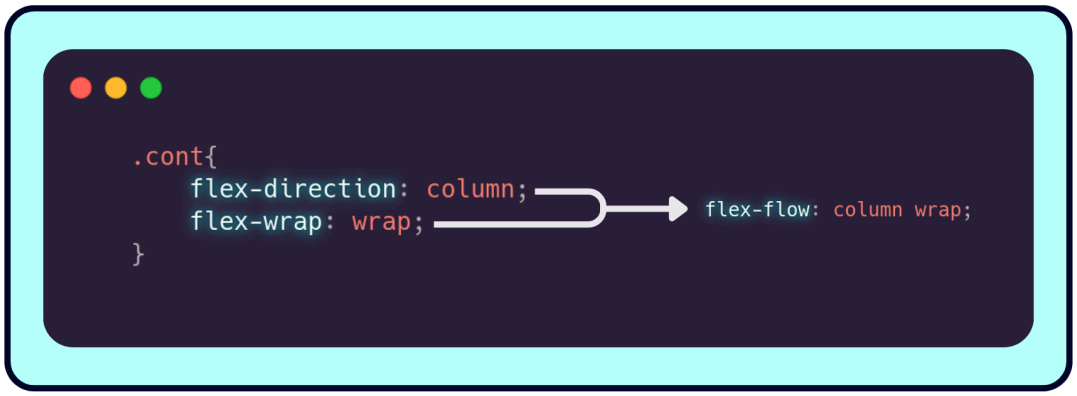
If you want to check it live, here is the CodePen link. Or scan this QR code:

The next property we're going to discuss is a shorthand.

Flex-flow Property

It's a shorthand for declaring ***flex-direction*** and ***flex-wrap*** properties together.

Like this:

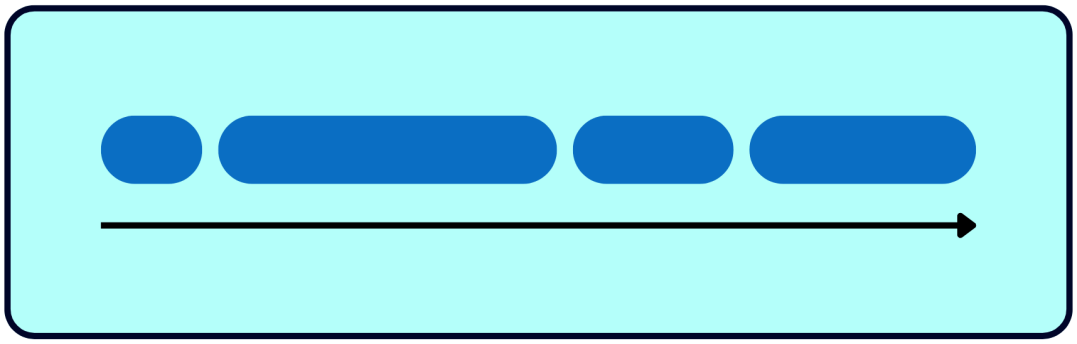


It takes two values in this order: ***flex-flow: <flex-direction> <flex-wrap>***. Both are separated by a space.

So, it can take all the possible combinations of these two properties.

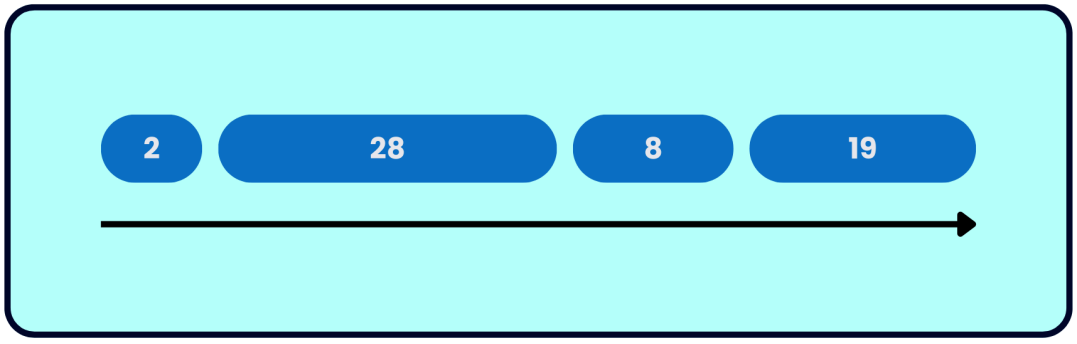
Order Property

Imagine we have a Flexbox with 4 flex items like this:

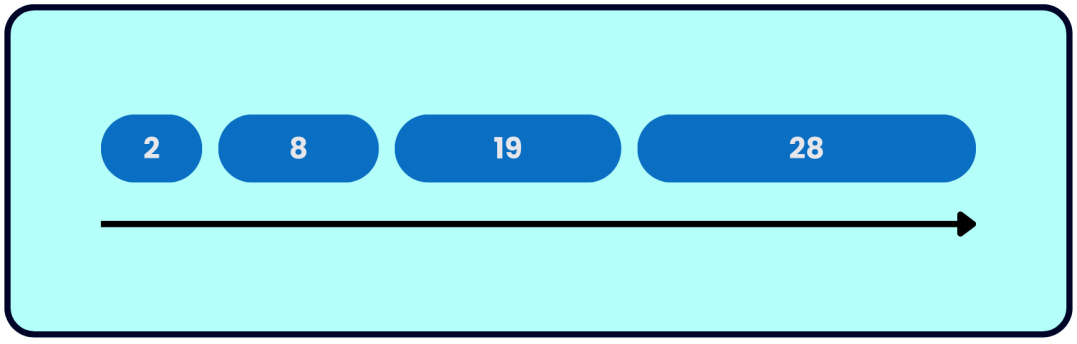


Now we assign a number to all 4 of these. What number?

Any number. Say, these:



Now Flexbox would reorder these items in the increasing order of these numbers. This way:



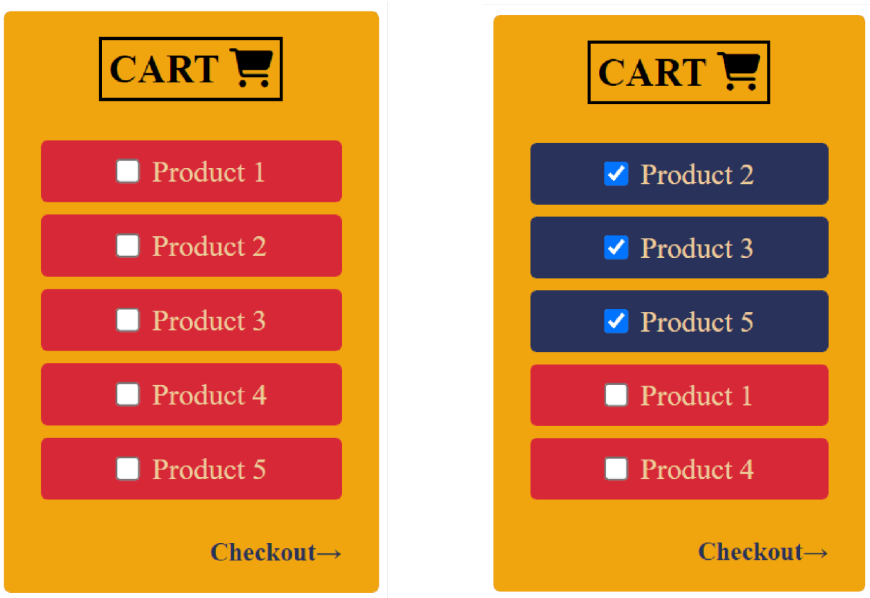
That's exactly what the order property does.

The default value is 0, so equal for all. Thus, flex orders them based on the markup.

But if you change it for any flex item, it'd be rearranged in increasing order.

Here's a cool use case:

Say, we have a cart layout like this:



We want all the checked items to move above the unchecked items. How'd you do that?

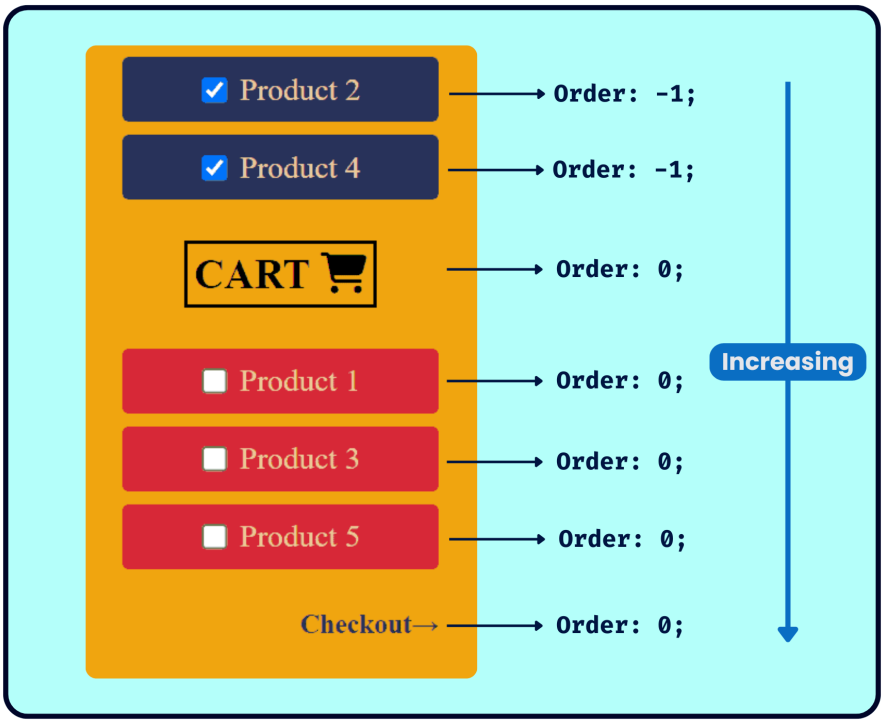
First, make the cart a flex container with ***flex-direction: column***.

Then simply set the checked item's ***order*** value to the least of all, so it comes to the top (e.g. -1).

Like this:

```
.cart-product:checked{

    order: -1;

}
```

But we have a little problem:



The checked products are going above the heading because they have an ***order*** of -1, while the heading has 0.

To fix this, set the heading's ***order*** to a lower value, like -2.

And we'll have a cart that changes the order of items when they're selected.

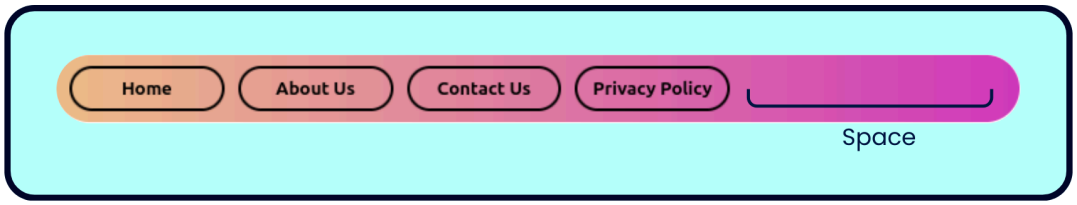
If you want to check it live, here is the CodePen link. Or scan this QR code:

Flex-grow Property

We know flex shrinks the items if it runs out of space and no ***flex-wrap*** is set.

But what if there is some extra space available?

Normally, it keeps it as it is. Like this:

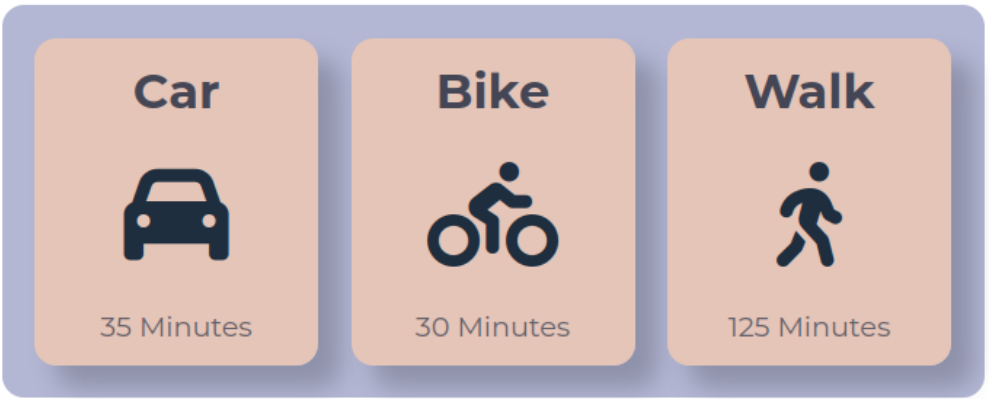


The ***flex-grow*** property lets you control if flex items should grow to fill available space and by what rate.

By default, it's set to 0, meaning no items grow.

You can change it, and the item will grow by that factor.

For instance, how'd you build this:



I'm not talking about the cards' alignment within the container, which we can do by placing all cards in a horizontal flex container.

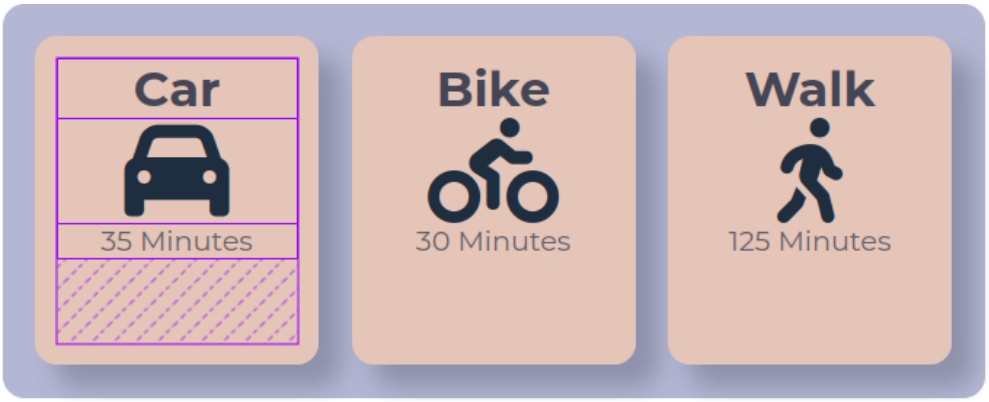
What about the items inside each card?

Each card has three elements like this:

```
<div class="card">
    <h3 class="name">Car</h3>
    <i class="icons"></i>
    <div class="time">35 Minutes</div>
 </div>
```

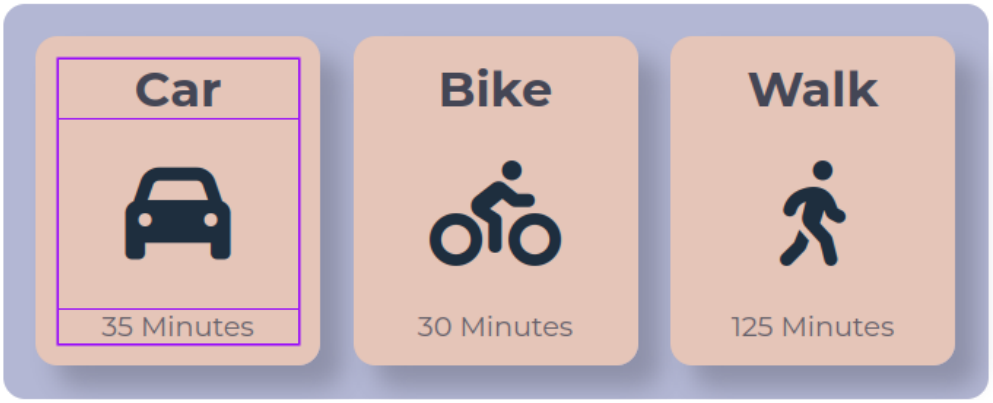
At first, you might think it's as simple as making them flex containers with ***flex-direction: column***.

But then you get this:



They're grouped at the top of the card, and the rest of the space is empty.

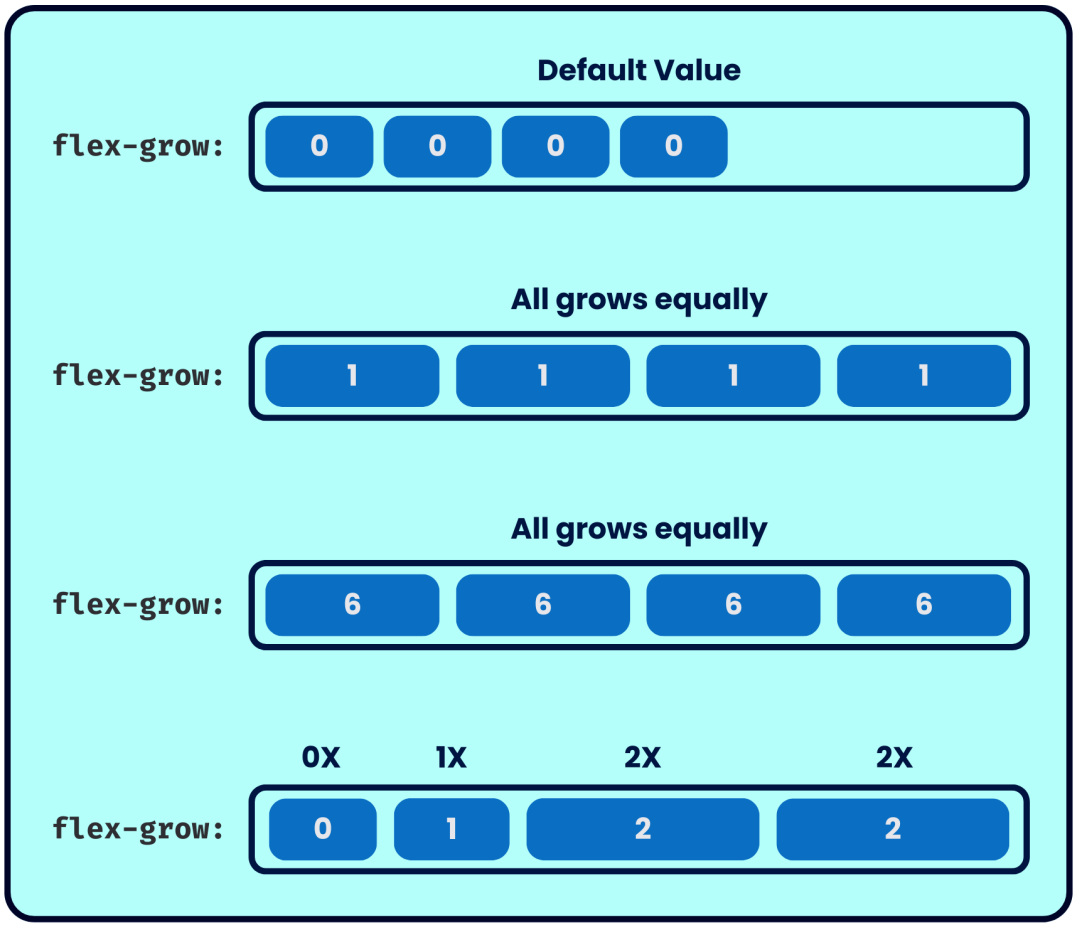
But if we set the ***flex-grow*** value of the icons to 1, then we get our original layout:



All the other elements' ***flex-grow*** is 0. So icons are expanding to cover up all the extra space.

Simply put, ***flex-grow*** grows the item with the given factor to occupy all the space.

Like this:



If you want to see the card example live, here is the CodePen link.

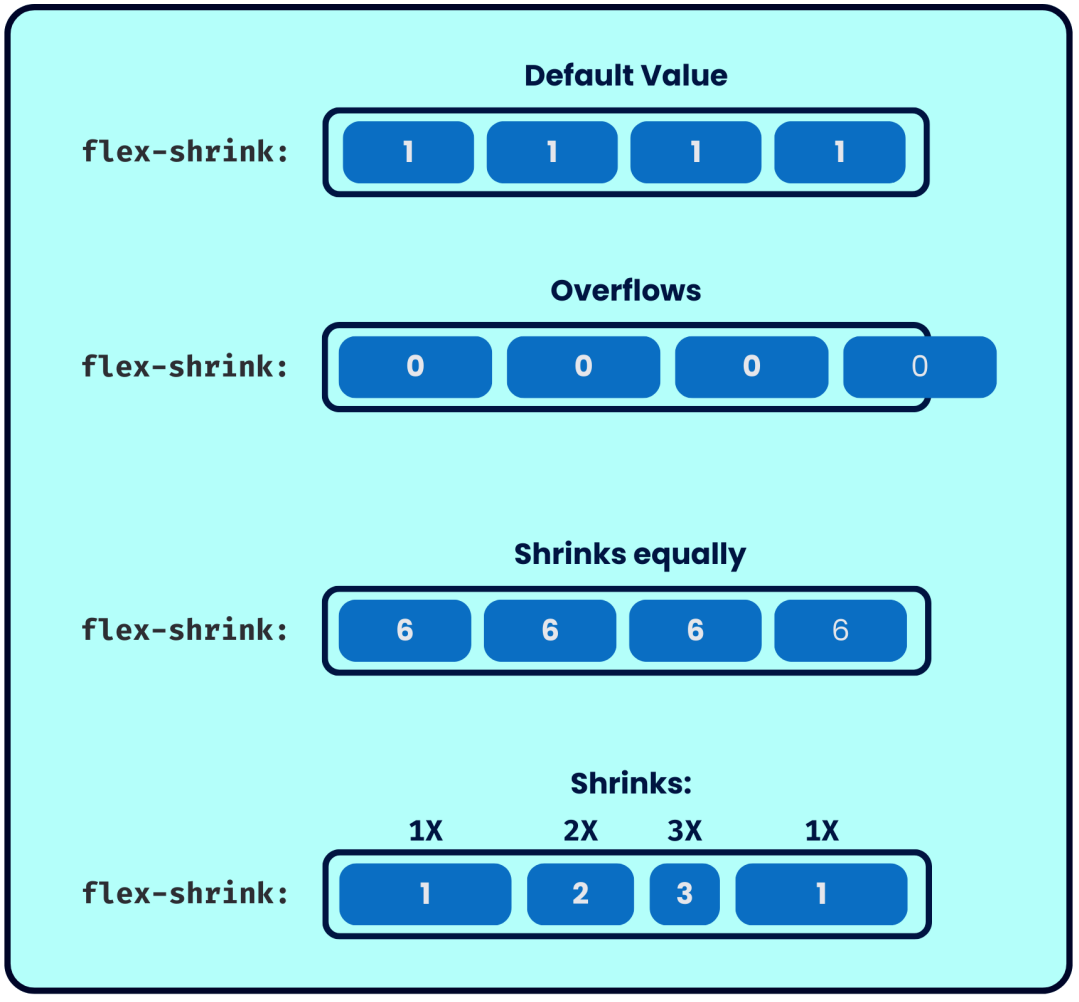
Or scan this QR code:

Flex-shrink Property

It's almost like ***flex-grow***, but it decides how much flex items should shrink when running out of space.

The default for it is 1. That's the reason flex items can shrink by default but never grow.

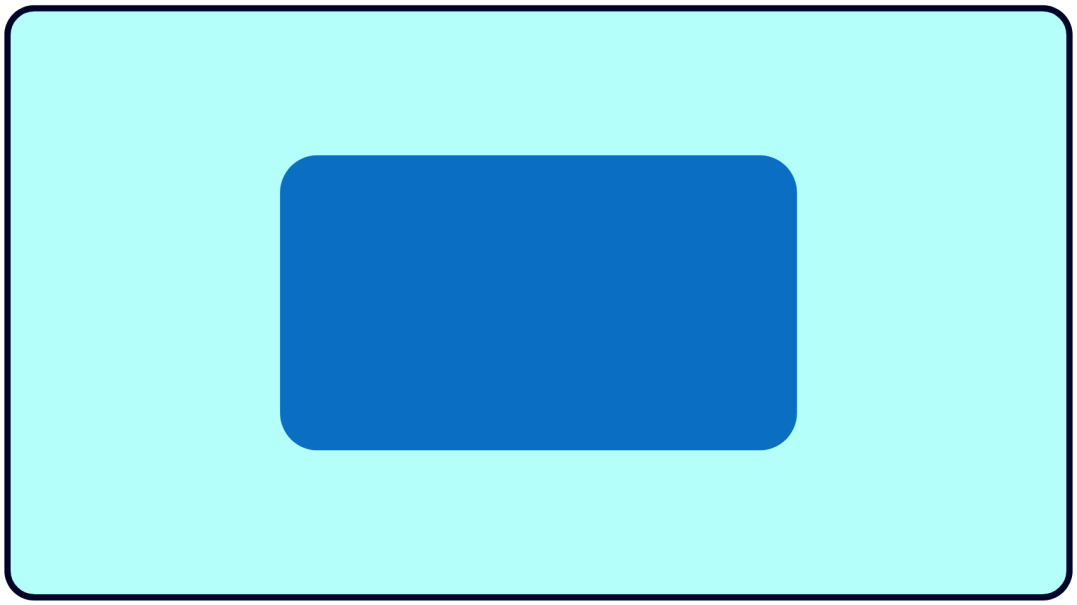
But if you want, you can make some elements shrink faster or some not shrink at all. Like this:



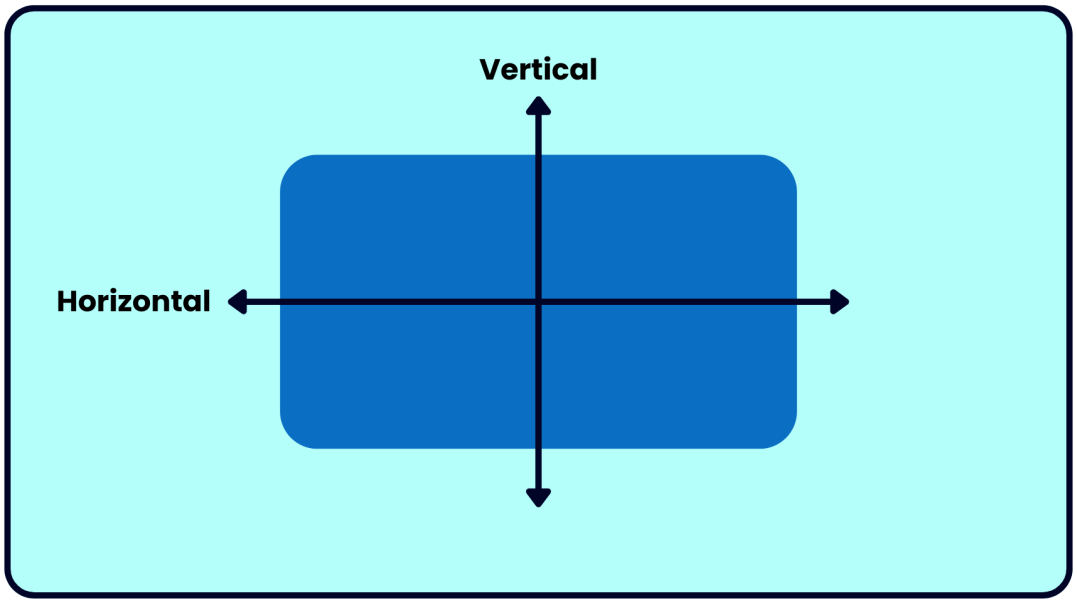
Before moving on to the next property, you must understand the basic flex structure.

Flexbox Structure

Until now, we've considered Flexbox as a box. So, let's start with that:



Every box has two axes: horizontal and vertical.

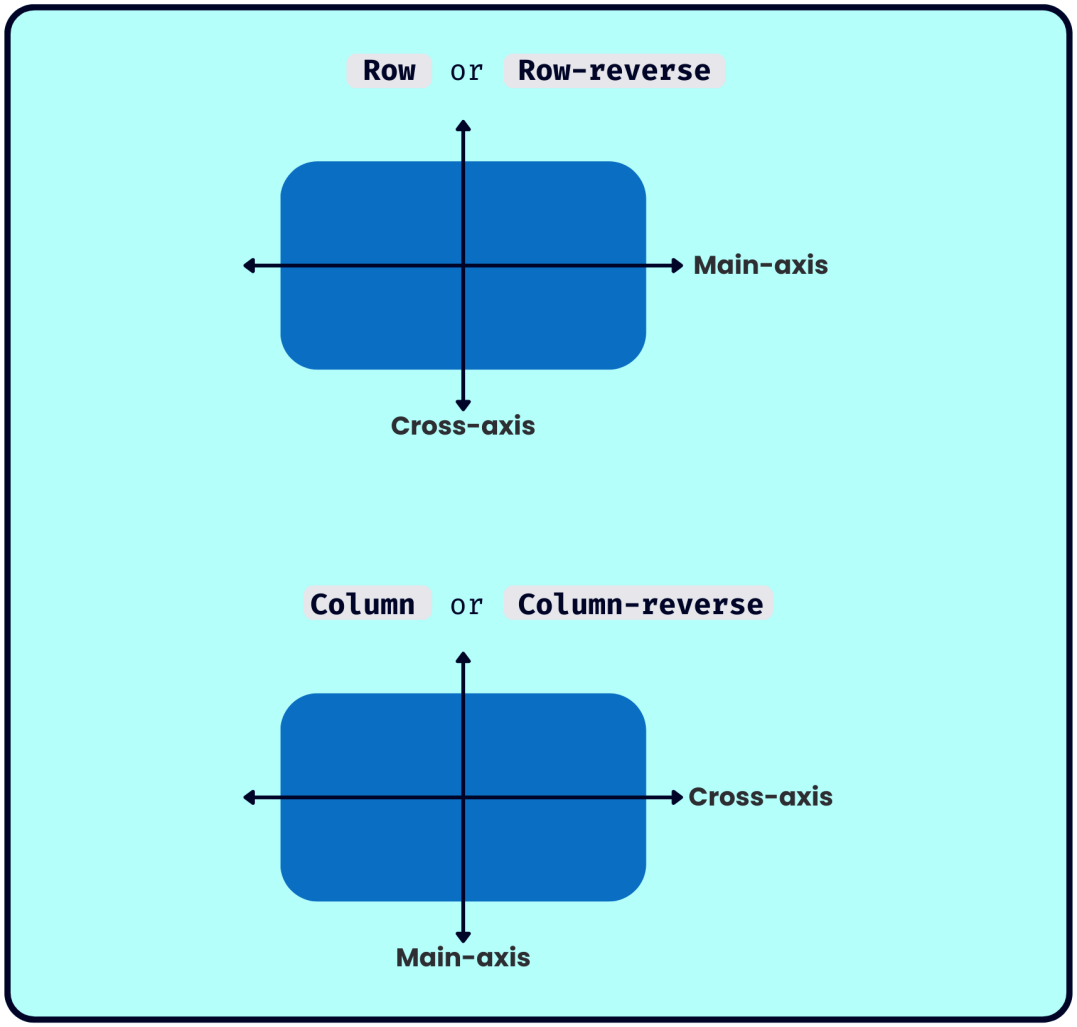


In Flexbox, we make one of these axes the ***main axis*** and the other one the ***cross axis***. How?

Well, using the ***flex-direction*** property. Do you remember? It can take 4 values:

Horizontal → *row* and *row-reverse*
Vertical → *column* and *column-reverse*

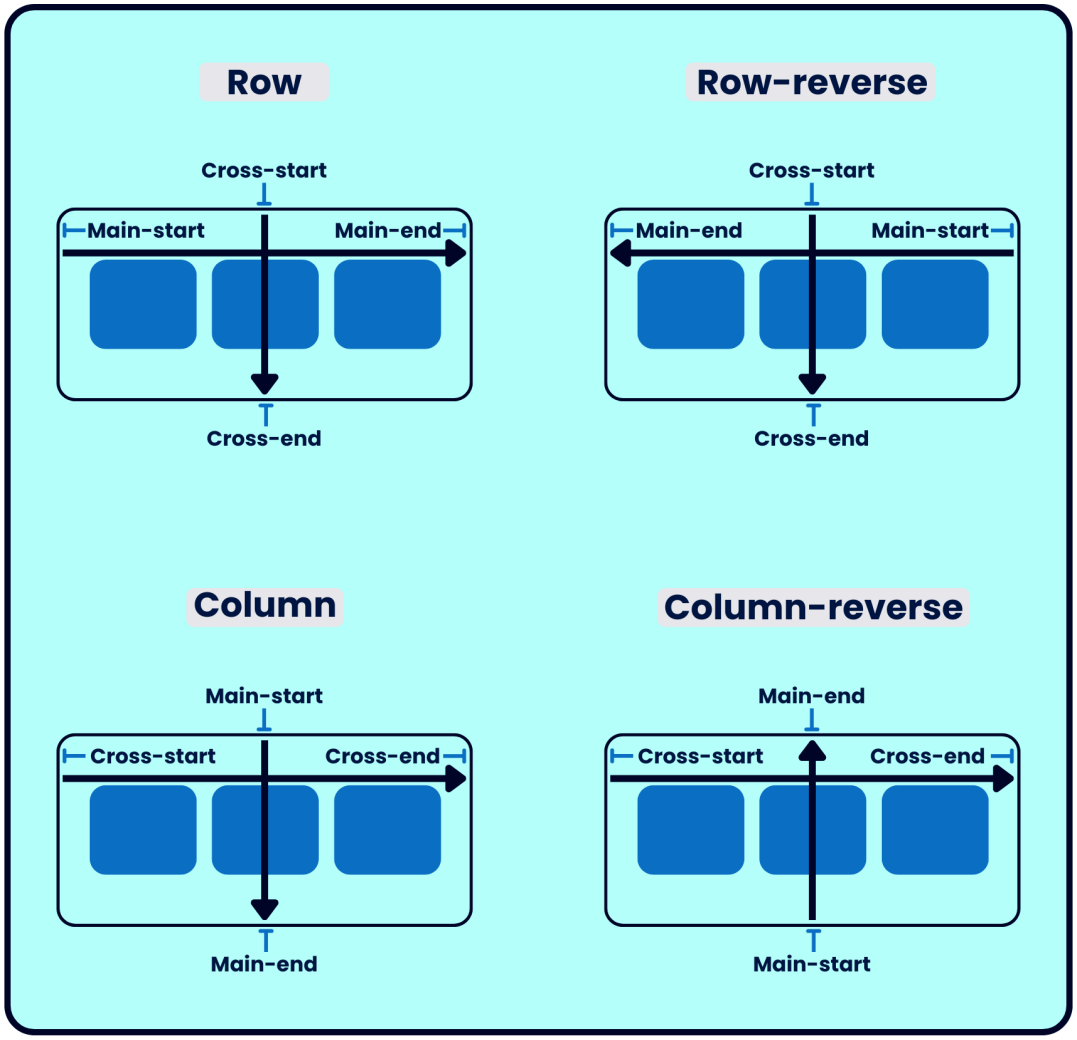
As you'd have guessed when ***flex-direction*** is horizontal, the **main axis** is horizontal; else, it is vertical. Like this:



But how do we differentiate between horizontal (*row* and *row-reverse*) or vertical (*column* and *column-reverse*)?

We know that even though both *row* and *row-reverse* are horizontal, one goes from left to right while the other right to left.

That's how we differentiate: by directions. Represented by ***main-start*** and ***main-end*** for *main axis*, ***cross-start***, and ***cross end*** for *cross axis*. Like this:



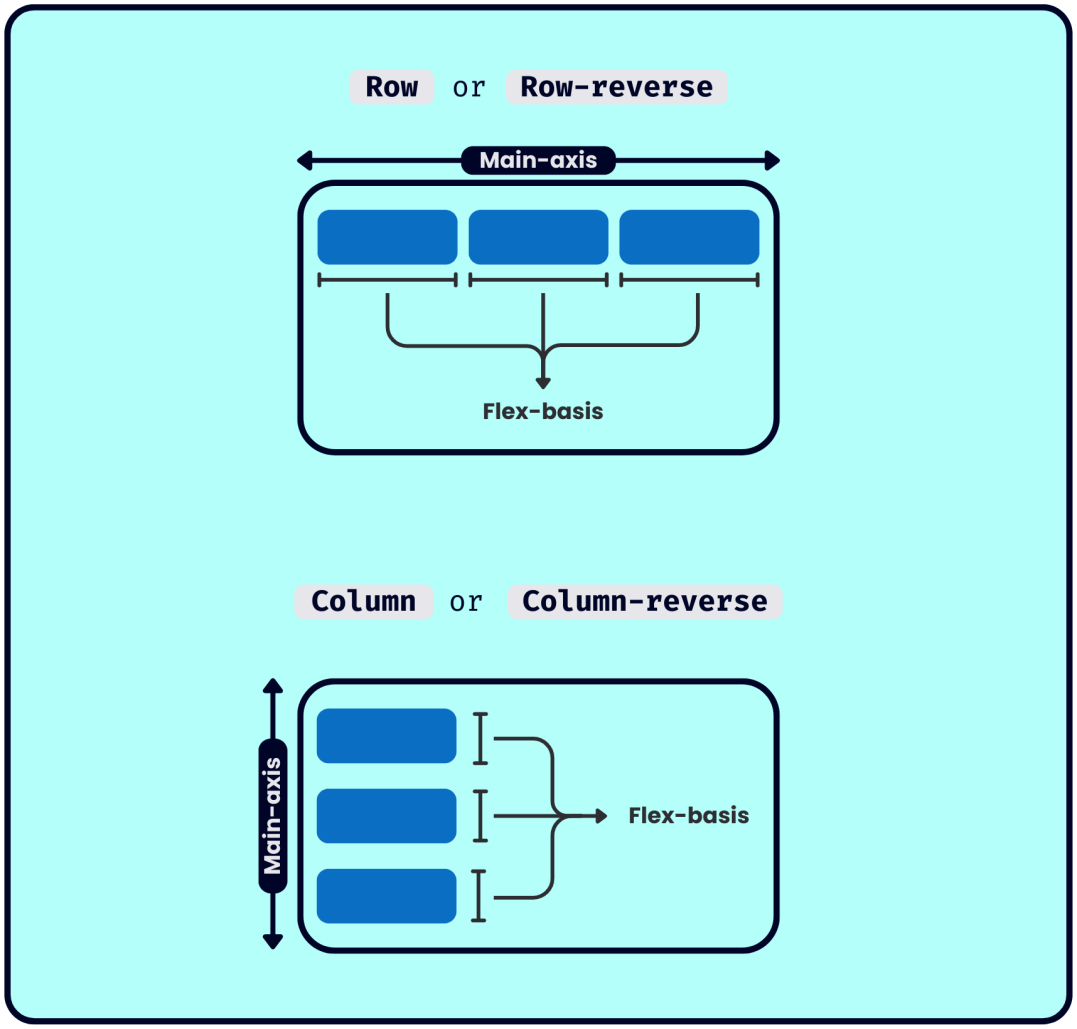
The terms might sound complex, but the idea is simple:

1. Check the container's ***flex-direction***. That's your *main axis*:
    a. For *row-reverse*, the *main axis* is horizontal, starting (*main-start*) from the right and ending (*main-end*) at the left.
    b. The *cross axis* is perpendicular to the *main axis*, with *cross-start* and *cross-end* in the usual direction (left to right or top to bottom).

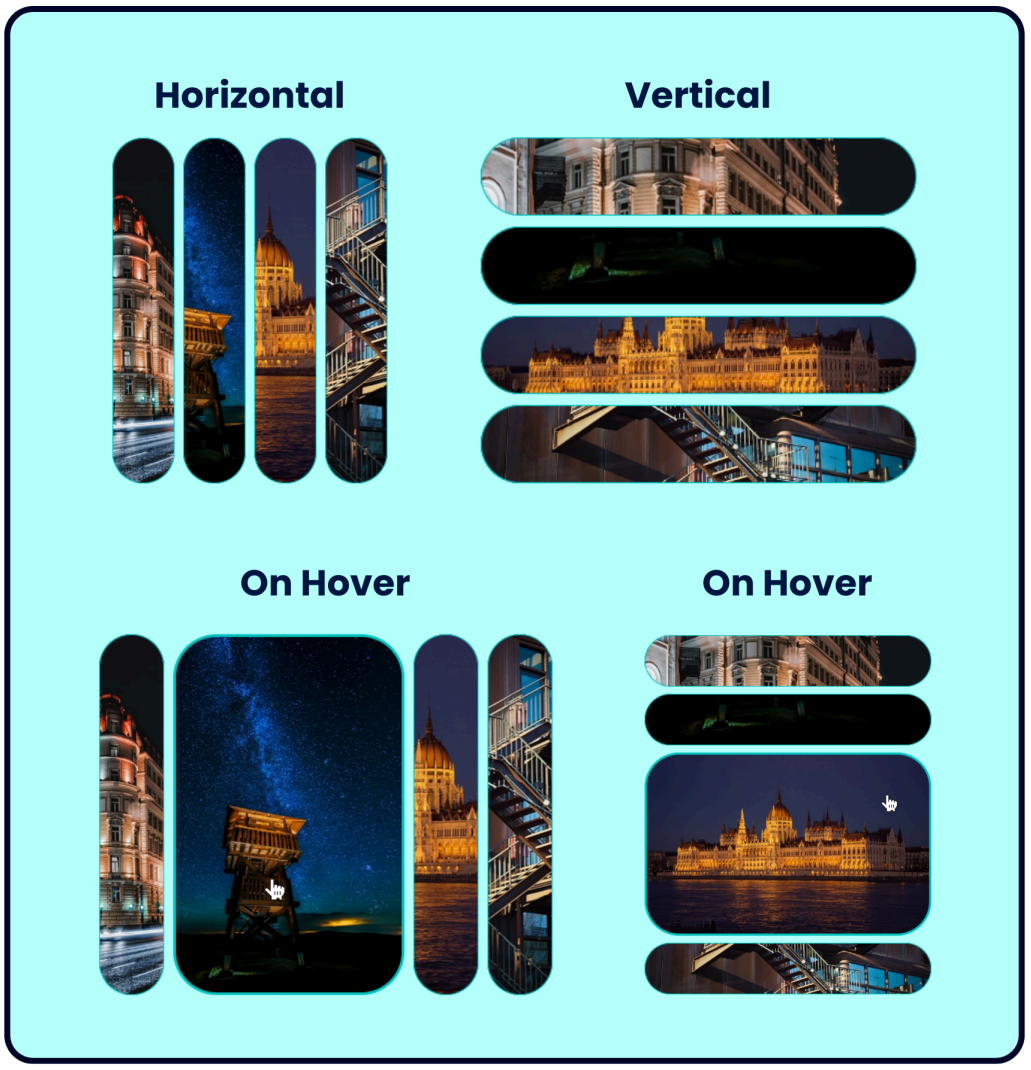
Got it? Great! Let's move on to the next property.

Flex-basis Property

A flex item has two sizes: width and height. So, the size of the flex item that's on the *main axis* is its ***flex-basis***. Like this:



Let's take this example:



On wide screens, the width expands; on narrow screens, the height expands.

We can switch ***flex-direction*** between *row* and *column* with a media query:

```
.cont {
    display: flex;
    flex-direction: column;
}
@media screen and (min-width: 480px) {
    .cont {
        flex-direction: row;
        height: 400px;
```

```
        }
    }
```

Normally, you'd also set/reset width and height and animate them individually with a media query.

But with ***flex-basis***, you can achieve all this simply by using:

```
.img{
    flex-basis: 70px;
}
.img:hover{
    flex-basis: 250px;
}
```

It automatically switches between width and height based on the ***flex-direction***.

> **Note** 📝
>
> If you specify both width/height and *flex-basis*, *flex-basis* will be given priority.

If you want to fiddle with the above example, here is the CodePen link. Or scan this QR code:

Now let's move on to a super cool shorthand property.

The Flex Shorthand

So far, we've covered:

flex-grow: Controls how items grow to fill space.
flex-shrink: Controls how items shrink when space is tight.
flex-basis: Sets the item's size on the main axis.

The ***flex*** shorthand combines all three. Say, you define these three properties:

```
.flex-item {
    flex-grow: 2;
    flex-shrink: 3;
    flex-basis: 200px;
}
```

Instead of that, you can use the ***flex*** shorthand like this:

```
.flex-item {
    flex: 2 3 200px;
}
```

It takes three values in this order: ***flex: <flex-grow> <flex-shrink> <flex-basis>***

But you can also give two or even a single value. Here's how it works:

```
/* One value, unitless: flex-grow*/
flex: 2; /* 2 1 0% */


/* One value, width/height: flex-basis */
```

```
flex: 10em; /* 1 1 10em */


/* Two values: flex-grow | flex-basis */

flex: 1 30px; /* 1 1 30px */


/* Two values: flex-grow | flex-shrink */

flex: 2 2; /* 2 2 0% */


/* Three values: flex-grow | flex-shrink | flex-basis */

flex: 2 2 10%;
```

It also accepts three keyword-based values:

```


flex: initial; /* Don't grow; shrink equally*/

flex: auto; /* Grow equally, shrink equally*/

flex: none; /* Don't grow, Don't shrink */
```

Now you can easily control the flex container and its items. Lastly, let’s learn to control alignment.

Justify-content Property

Remember 🔔

In CSS, justify-* controls alignment on the main axis, and align-* on the cross axis.

Do you remember the navbar example from the start? This one:

We moved the buttons to the right with ***flex-direction: row-reverse***, which also changed the flex structure.

That's why *"Home"* is on the right, even though it's first in the markup.

Justify-content lets you align flex items along the *main axis* without changing the structure.

If in the above case, we'd have used this:

```
nav{
    display: flex;
    flex-direction: row; /* Default */
    justify-content: flex-end;
}
```

Then it'd become this:

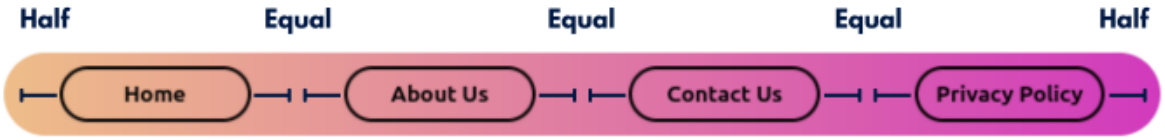
See the difference? The order is intact this time.

It accepts six values:

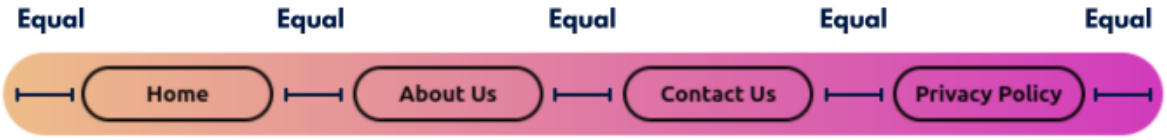
1. ***flex-start***: Groups items at the *main-start* side (default).
2. ***flex-end***: Moves items to the *main-end* side.
3. ***center***: Centers the items, like this:

4. ***space-between***: The space is put **between** all consecutive flex items:

5. ***space-around***: Puts the space **around** all the flex items. Like this:



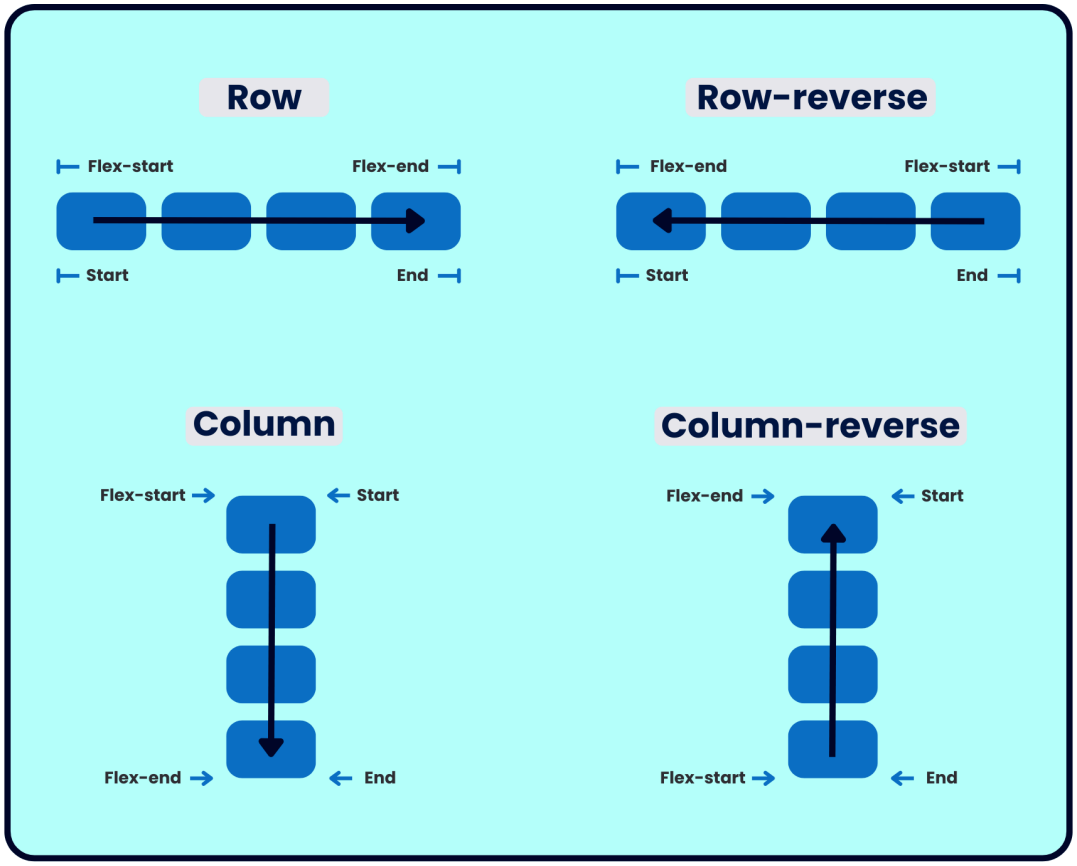
6. ***space-evenly***: Space the items equally from each other as well as the edges:



Note 📝

You can also use *"start"* and *"end"* instead of using them with the ***flex*** prefix, but they're general box model values and ignore the flex structure.

So, start always means left or top, and end means right or bottom, like this:



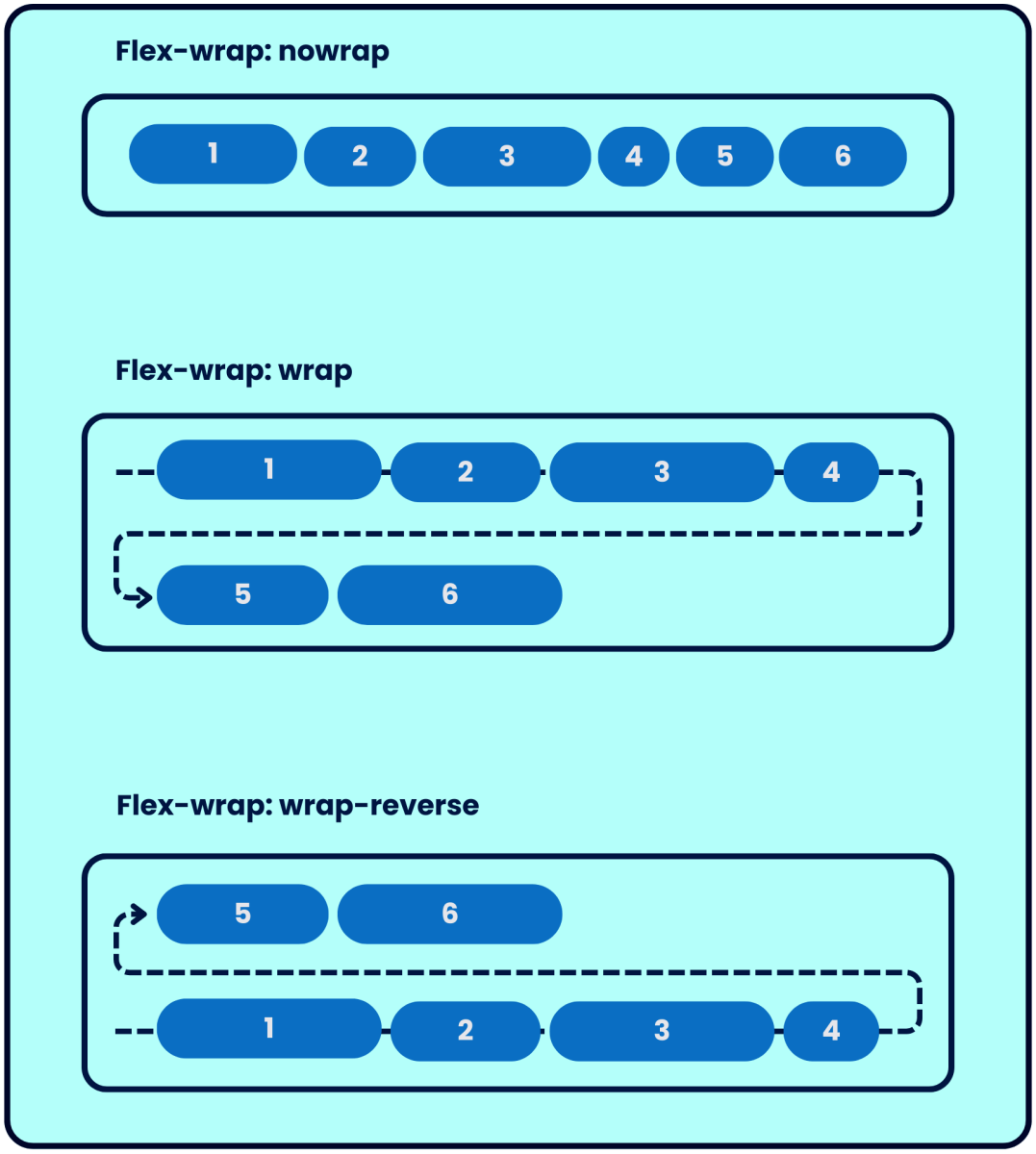
If you want to check this updated navbar, here is the CodePen link. Or scan this QR code:

So far, we've discussed aligning items on the main axis. What about the cross-axis?

Let's find out…

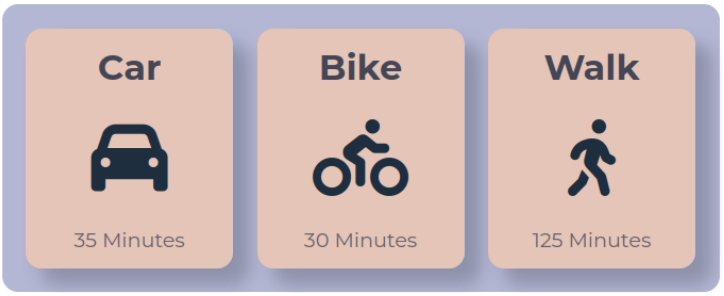
Align-content Property

Do you remember what the ***flex-wrap*** property does? Just to recap:

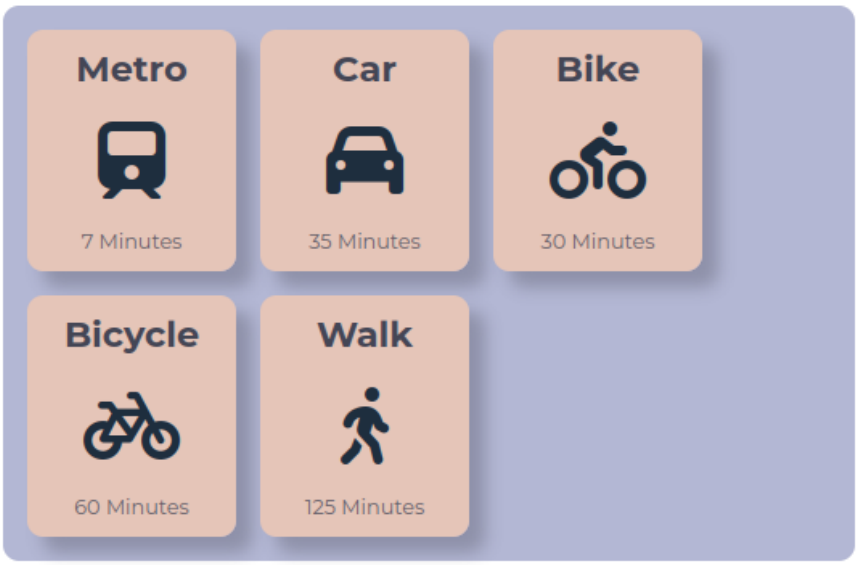


The ***align-content*** property lets you control the alignment of these individual *rows/columns* along the *cross-axis*.

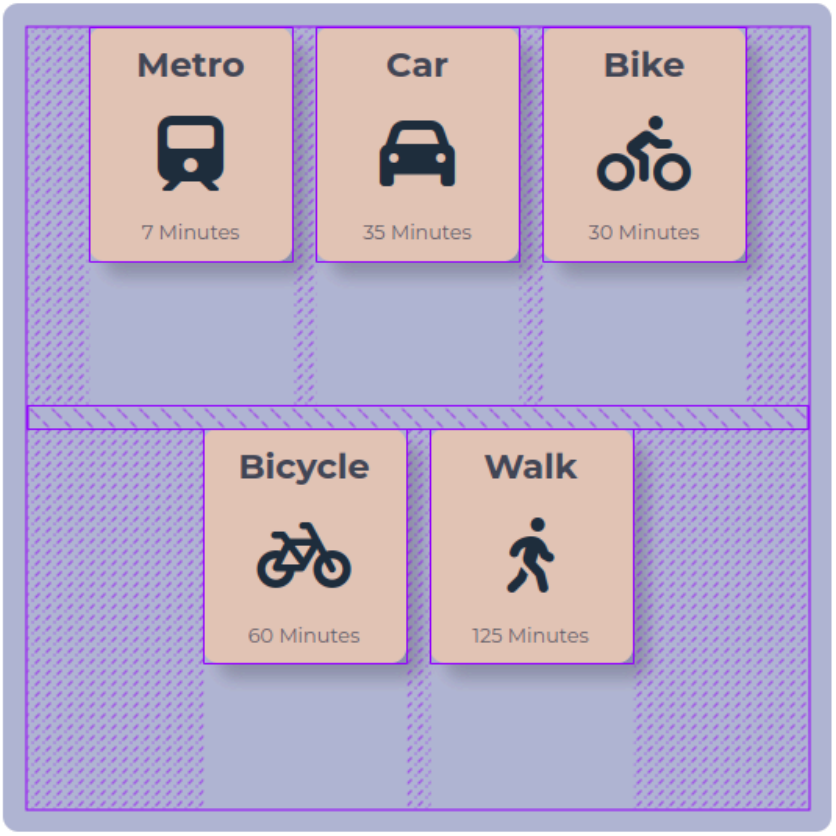
You'll get a better idea with an example. Remember this?



What if we add some more cards to it? It'd then wrap the Flexbox as we have ***flex-wrap: wrap***:

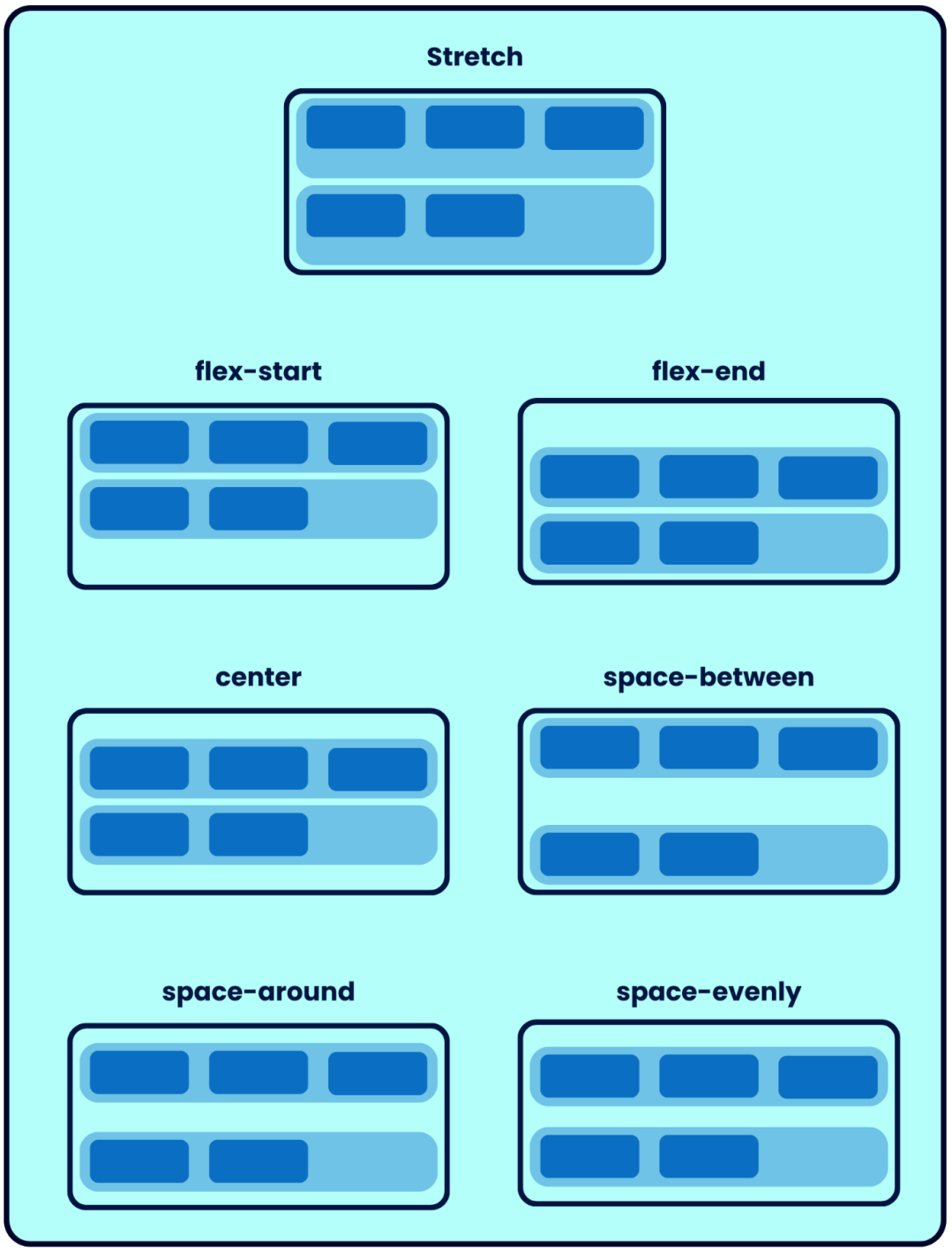


To align the cards horizontally, we can use ***justify-content: center***. But if the main container has a fixed height, like this:



Each row is stretched to cover half the container's height because, by default, ***align-content*** is set to *stretch*, which makes the tracks fill the entire container.

You can change this with other values, like this:



So, if we set ***align-content: cente***r in our example, we'll get this:

Note 📝

This property controls the alignment of rows/columns, so it only works when *flex-wrap* is set to *wrap* or *wrap-reverse*.

If you want to check this example live, here is the CodePen link. Or scan this:

Place-content Shorthand

We learned ***justify-content*** and ***align-content***. If you're setting both of them to the same value. Say, this:

```
.cont{
    display: flex;
    justify-content: center;
    align-content: center;
}
```

You can use the ***place-conten***t shorthand to define them both at once, like this:

```
.cont{
    display: flex;
    place-content: center;
}
```

This will set both of them to center.

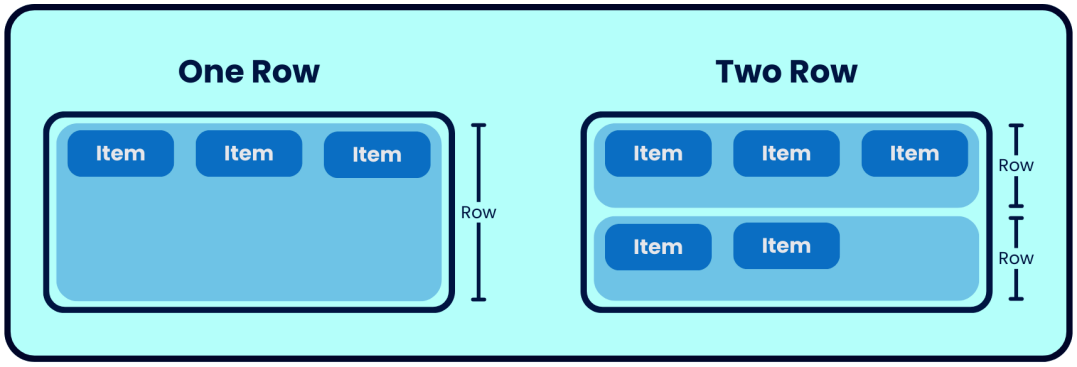
That's it. That's the use of this property.

In the last example, in stretch value, did you observe that the flex items are aligned at the top within the row?

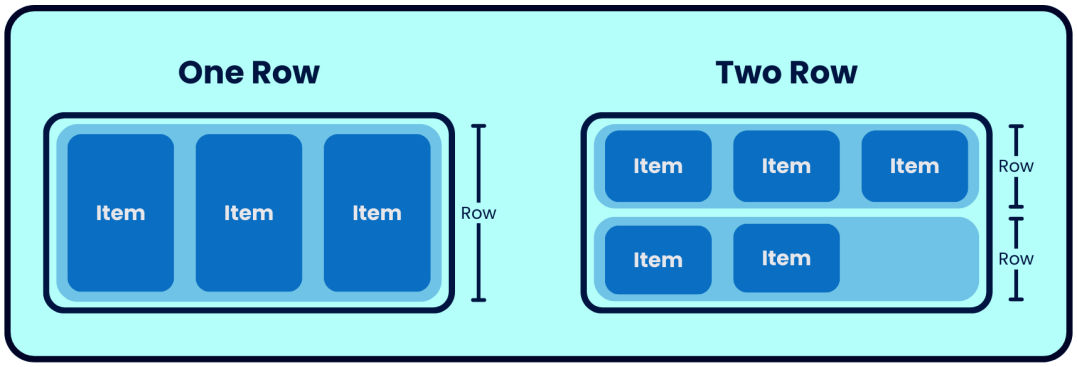
The next property lets you change that.

Align-items Property

The ***align-content*** lets you align the flex items within their tracks. So, normally, we have the items like this:



If we don't set a fixed height on them, they stretch to cover up the rows. Like this:

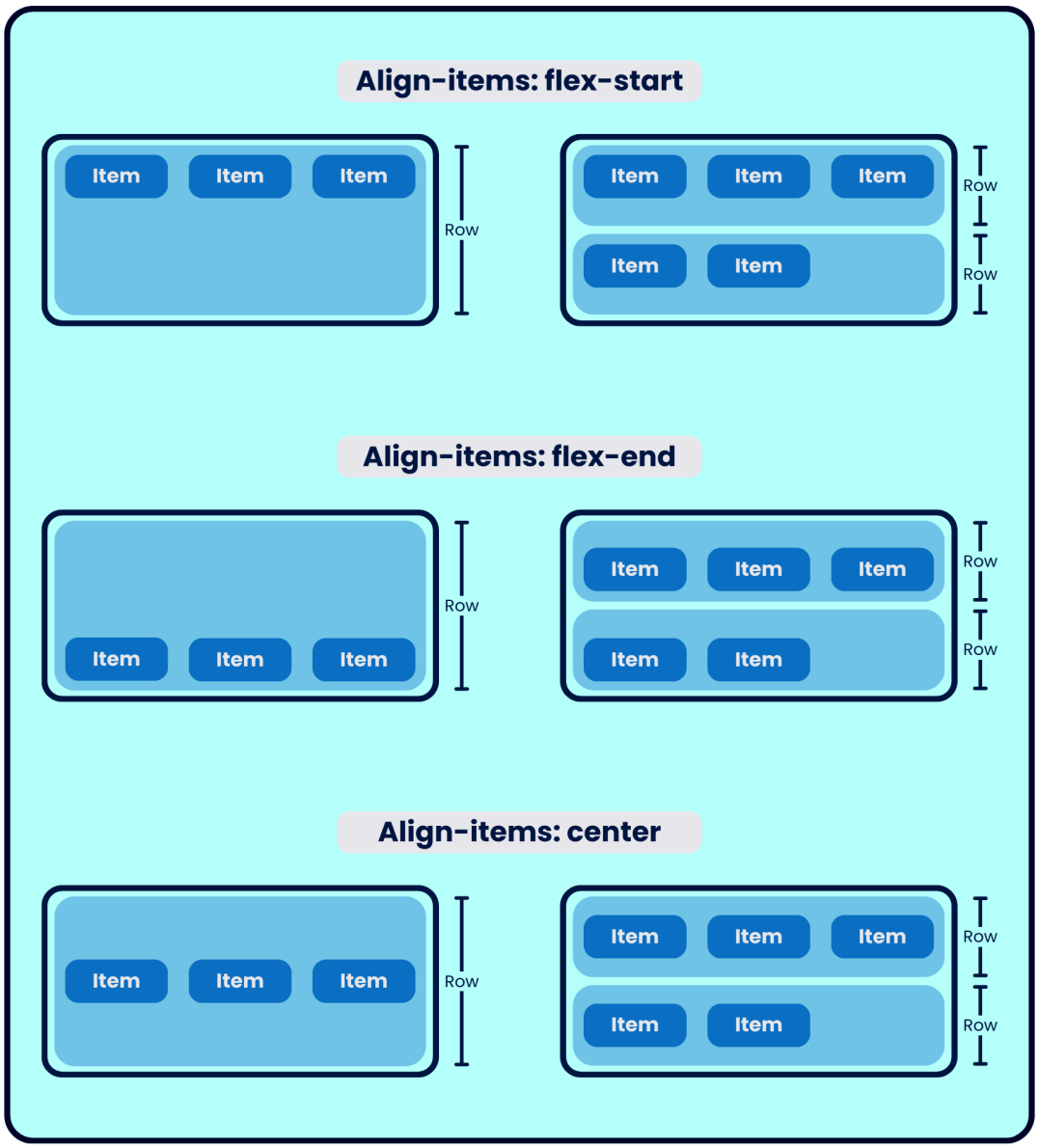


These items are stretching because ***align-items*** is set to stretch by default.

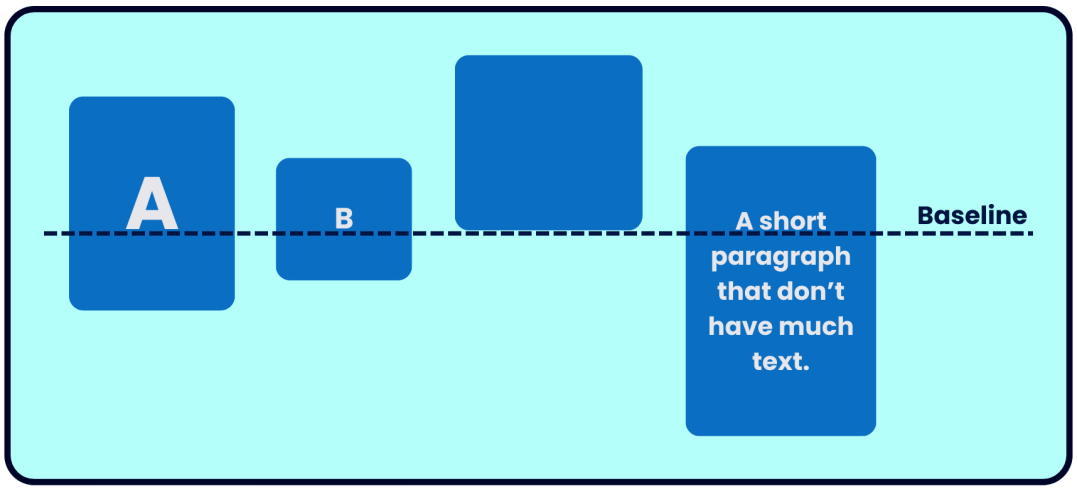
But you can change that with other values:

1. ***stretch***: Stretches items to fill the track (default).
2. ***flex-start***: Aligns items at the cross-start side.
3. ***flex-end***: Aligns items at the cross-end side.
4. ***center***: Centers items in the row/column.

Like this:



It takes another interesting value: *baseline*. But first, what's a baseline?



And that's exactly what the baseline value does. It aligns the flex items so all of their *baselines* match.

One of its good use cases is where we need to align a heading and subheading:

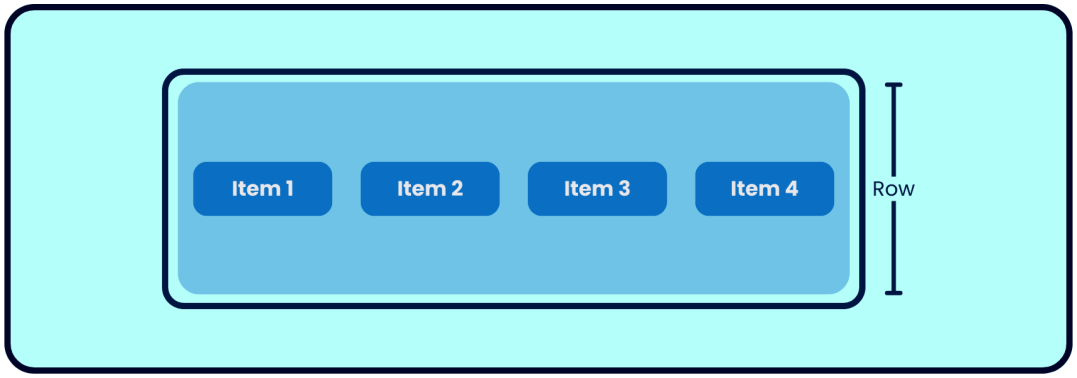
Align-self Property

If you know ***align-items***, you already know ***align-self***.

Align-self is just ***align-items***, but for specific items. So, if you do this:

```
.flex-cont {
  Display: flex;
  Align-items: center;
}
```

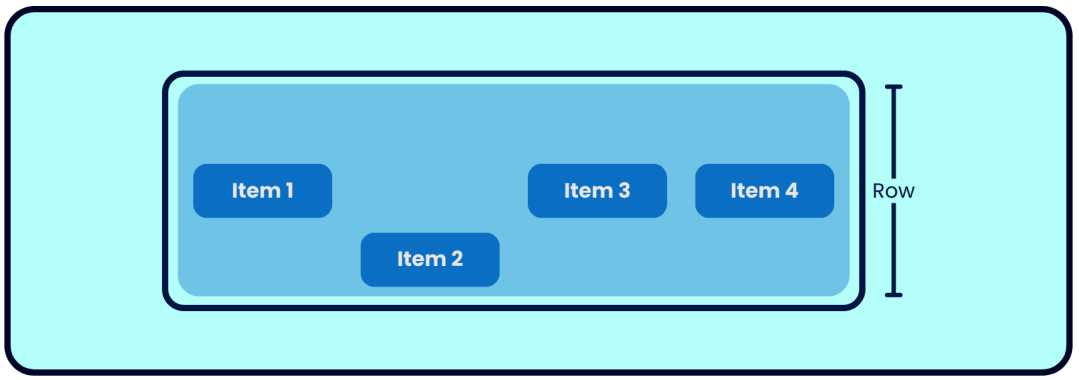
It centers **all** the flex items within their tracks, like this:



But now if you want to change the alignment of just item 2, you can overwrite its alignment like this:

```
.item2 {
    align-self: flex-end;
}
```

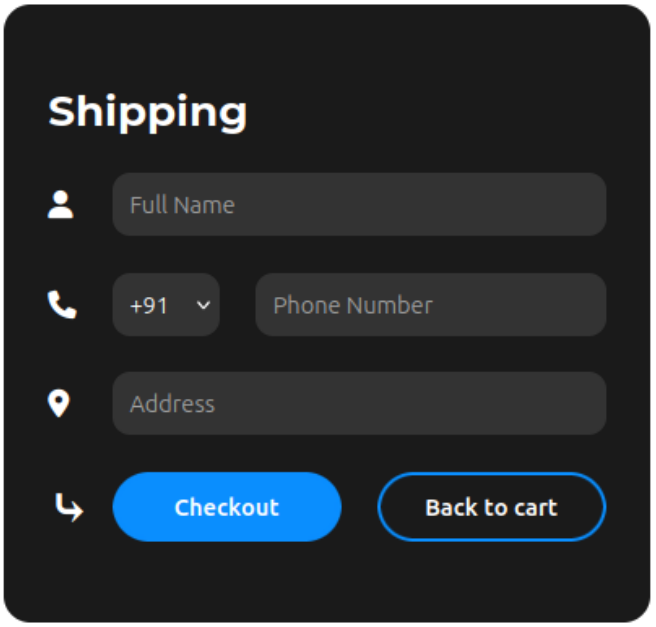
It'd now look like this:



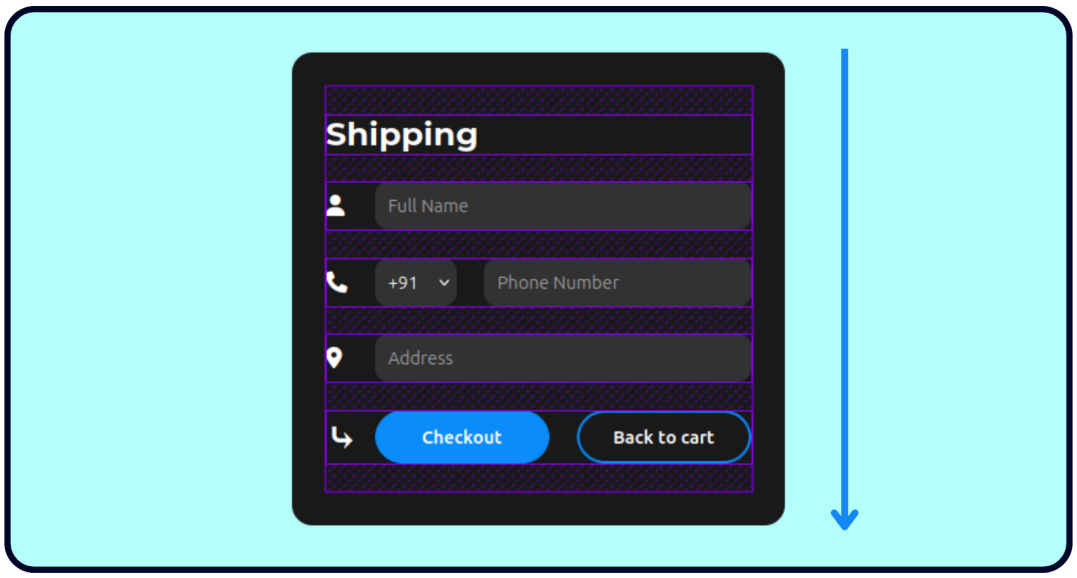
Align-self is used to overwrite the ***align-items*** value for a specific flex item.

Thus, it's defined for the **flex container** while ***align-self*** is defined for **flex item**.

Let's see this last example to clarify all the alignment properties:

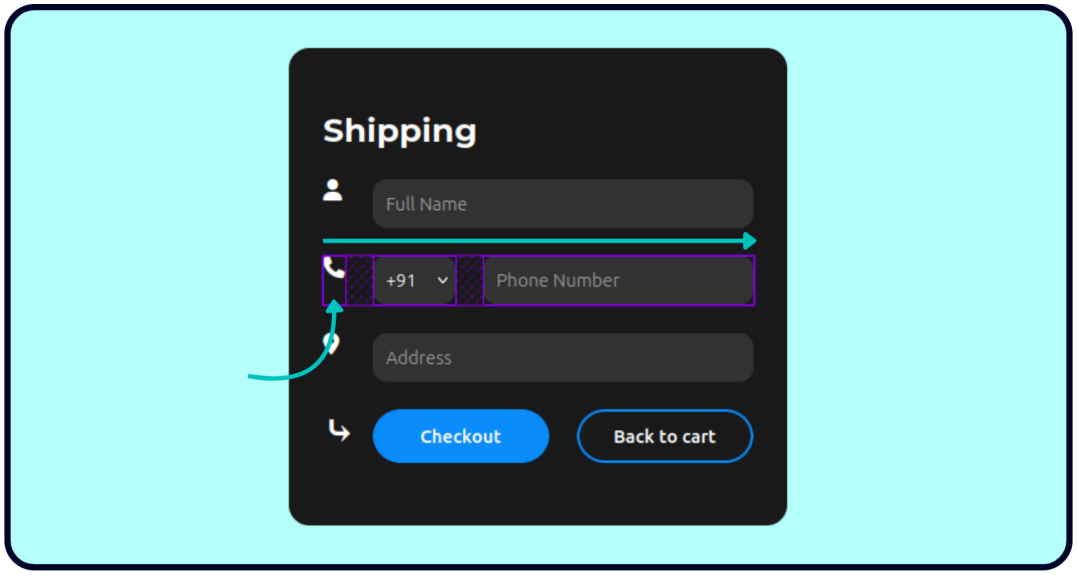


First, we wrapped each row in a wrapper. So, for the main container, each row acts as one flex item, like this:



We set ***flex-direction: column***, making the vertical axis the *main axis*, and used ***justify-content: space-evenly*** to evenly space the rows vertically.

Then, we made each row a horizontal flex container with ***flex-grow: 1*** for all items, so they grow. Like this:



As you can see, the icons aren't aligning with the inputs. Since it's the cross-axis, we'll do this: ***align-items: center***. And it'd place them perfectly.

Or, you can also use ***align-self***. Because we just need to align one flex item.

```
.form-row{
  display: flex;
  align-items: center; /* This */
}
.icon {
  align-self: center; /* Or this */
}
```

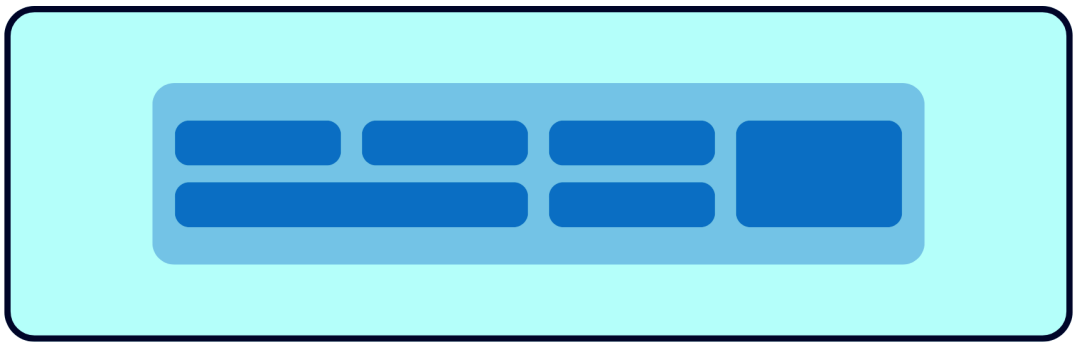
Once done, you'll get the perfect layout. Here is the CodePen link for it. Or scan this:

Grid

Why Grid Now?

With Flexbox, we can do almost anything—create layouts, control alignments, and adjust sizes.

Then what's the need for the grid? Well, check this layout:



In Flexbox, rows and columns aren't aware of each other, so we can't span items across tracks or overlap elements.

Or grid simply solves certain types of problems more efficiently.

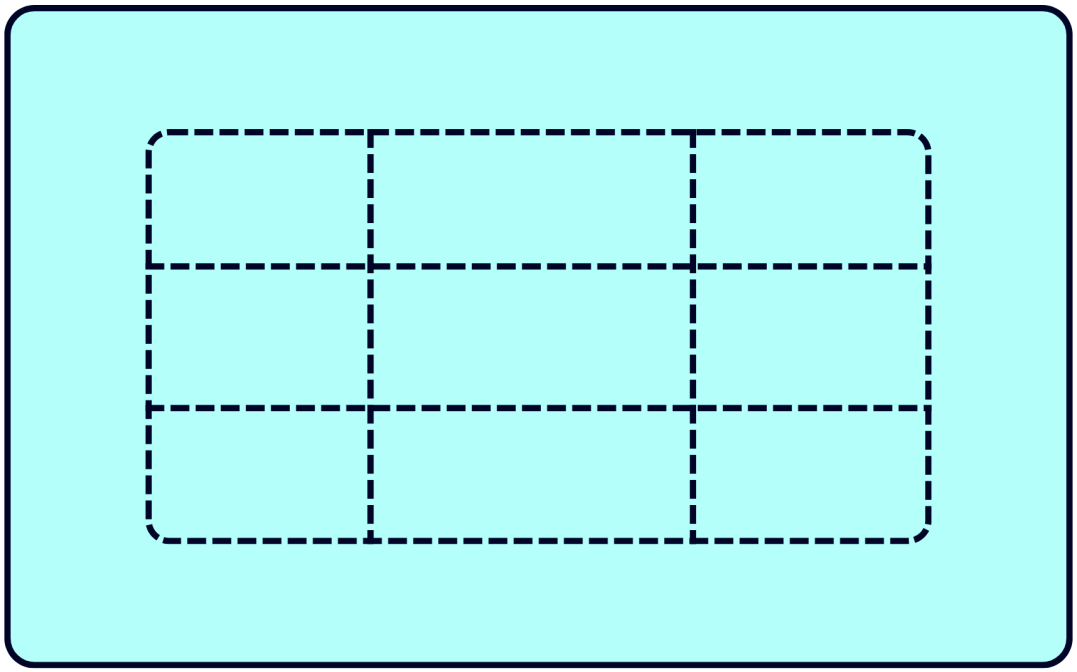
Let's see how it does that.

3 Steps to Grid

Here's the basic idea of the grid:

1. You make a grid container with ***display: grid*** or ***display: inline-grid***.

2. You declare a *structure/template* for grid with any number of rows and columns. Like this:



3. Finally, you place grid items in the cells—they can span rows, columns, or both, and even overlap. But let's first learn how to create a grid template.

Defining the Grid Template

Once you've made a container a grid container, you can use the first property.

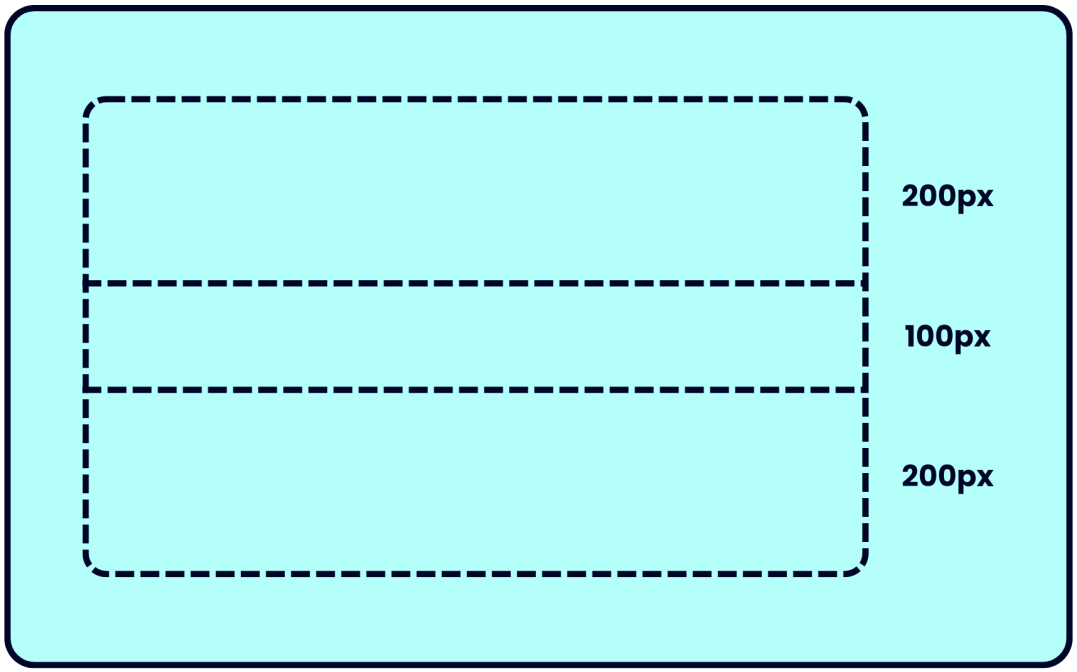
Grid-template-rows

As the name says, it lets you define the number and size of rows in the grid.

So, if you use this:

```
.container{
    display: grid;
    grid-template-rows: 200px 100px 200px;
}
```

It'd make the container a grid container with three rows, like this:



What about columns?

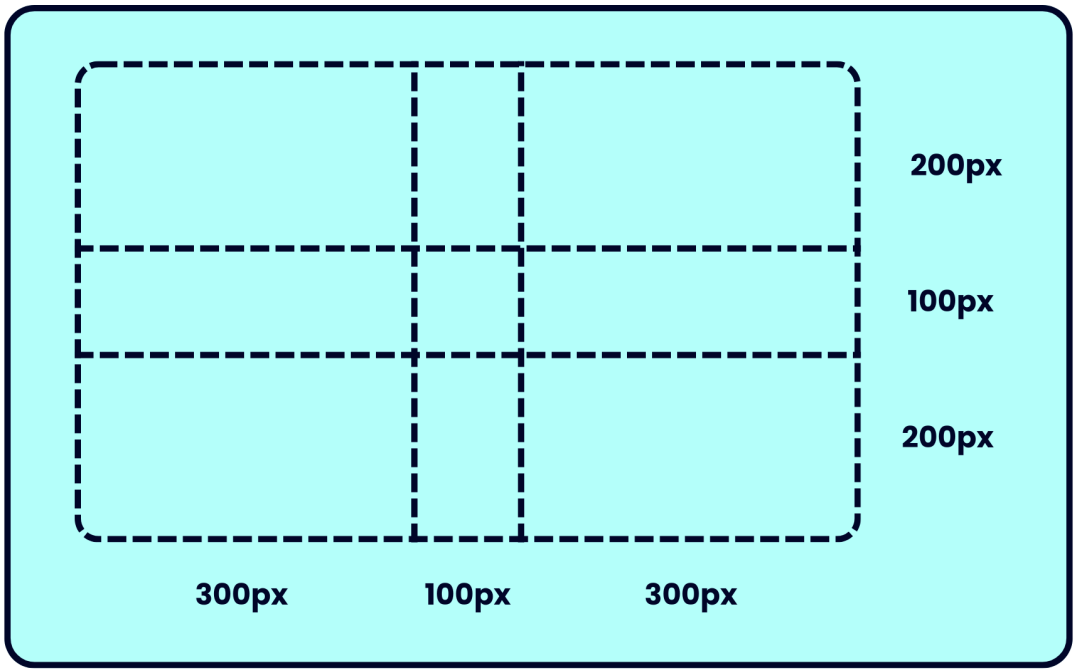
Here, it only created one column (default behavior), but you can change it with the next property.

Grid-template-columns

So, if you were to make a 3 by 3 grid, you can do so with this code:

```
.container{
    display: grid;
    grid-template-rows: 200px 100px 200px;
    grid-template-columns: 300px 100px 300px;
}
```

And you'd get this:



You're not limited to just using the px unit. You can use other sizing units as well:

1. em
2. rem

3. %
4. vw
5. max-content
6. Or any other.

But other than that, you can also use some other interesting values that are specific to these two properties. Let's discuss them.

Fractional Unit (fr)

As the name says, it creates tracks based on the fraction of the container.

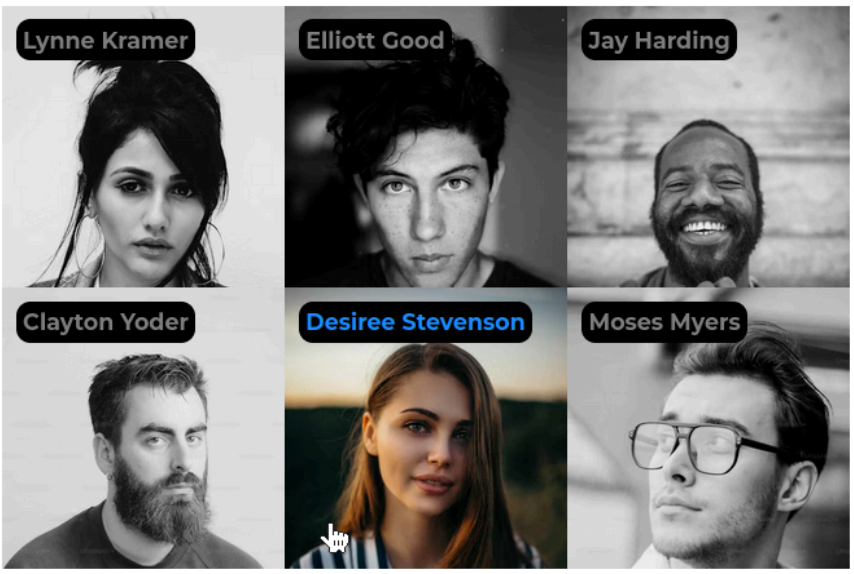
To create 3 equal columns that span the entire width:

```
.container{
    display: grid;
    grid-template-columns: 1fr 1fr 1fr;
}
```

Or if you want the middle row twice as the other two:

```
.container{
    display: grid;
    grid-template-rows: 1fr 2fr 1fr;
}
```

If you want to make a 3 x 2 image grid:



Then you can create the template like this:

```
.cont{
    display: grid;
    grid-template-rows: 1fr 1fr;
    grid-template-columns: 1fr 1fr 1fr;
}
```

Once the structure is set, each cell will by default display one image, as shown above.

Here's the CodePen link for it, or scan this QR code:

Minmax() Function

It takes two values: *min* and *max*, and then it could vary the size between these two values but never exceed them.

If you have this:

```
grid-template-columns: minmax(100px, 150px) 1fr;
```

Then the first column would be a minimum of 100px and a maximum of 150px but never cross this limit.

We will see an interesting use case for this in a while.

Repeat() Function

Again, as the name suggests, it lets you repeat a certain value instead of writing it multiple times.

For example, if you want to create a 12-column grid system, you'd need this:

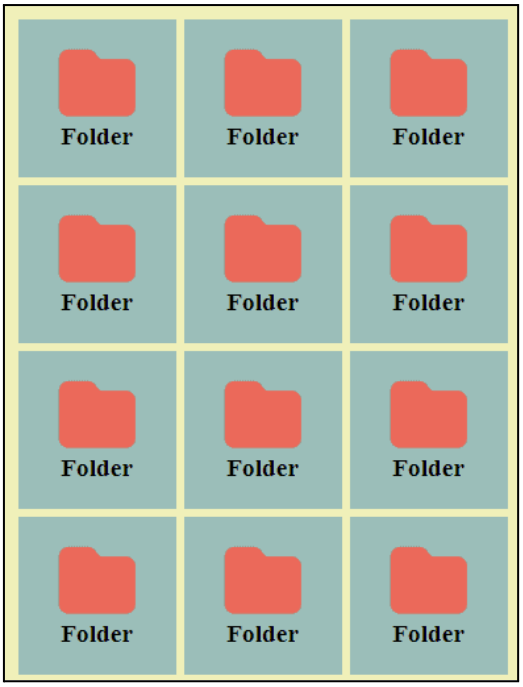
```
grid-template-columns: 1fr 1fr 1fr 1fr 1fr 1fr 1fr 1fr 1fr 1fr
                       1fr 1fr;
```

So, you can use the repeat function instead:

```
grid-template-columns: repeat(12, 1fr);
```

It takes two arguments. The first is the number of times to repeat, and the second is the size of those tracks.

Let's take an example:



It has three columns. So, we'd need this:

```
.cont {
    grid-template-rows: auto;
    grid-template-columns: 1fr 1fr 1fr;
}
```

Rows are set to auto by default, creating as many as needed, so we can omit that. For the columns, we can rewrite it as:

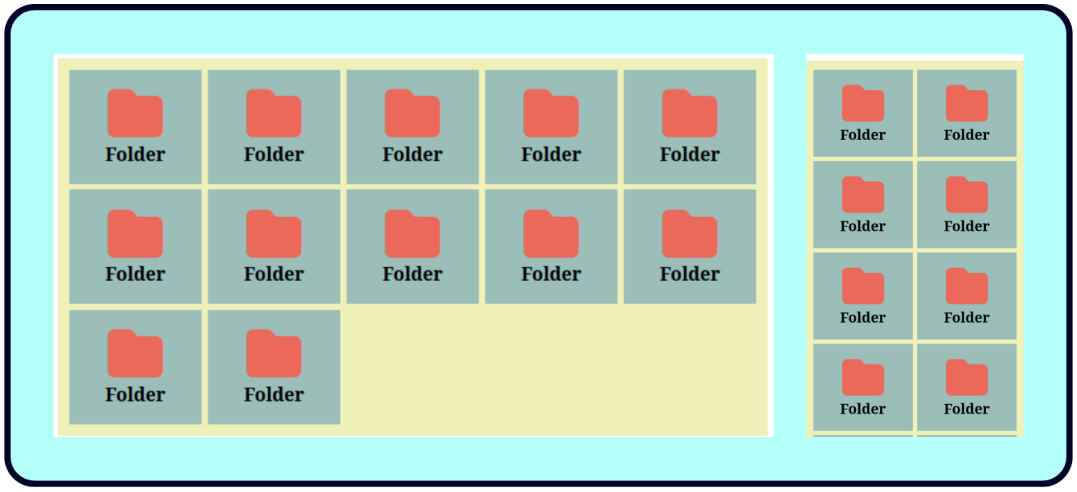
```
.cont{
    grid-template-columns: repeat(3, 1fr);
}
```

It'd create 3 columns of size 1fr.

Here, we've manually set the number of columns in the ***repeat()*** function. But you can also make it dynamic with the next values.

Auto-fit and auto-fill Values

If we want the number of columns to adjust based on the width, like this:



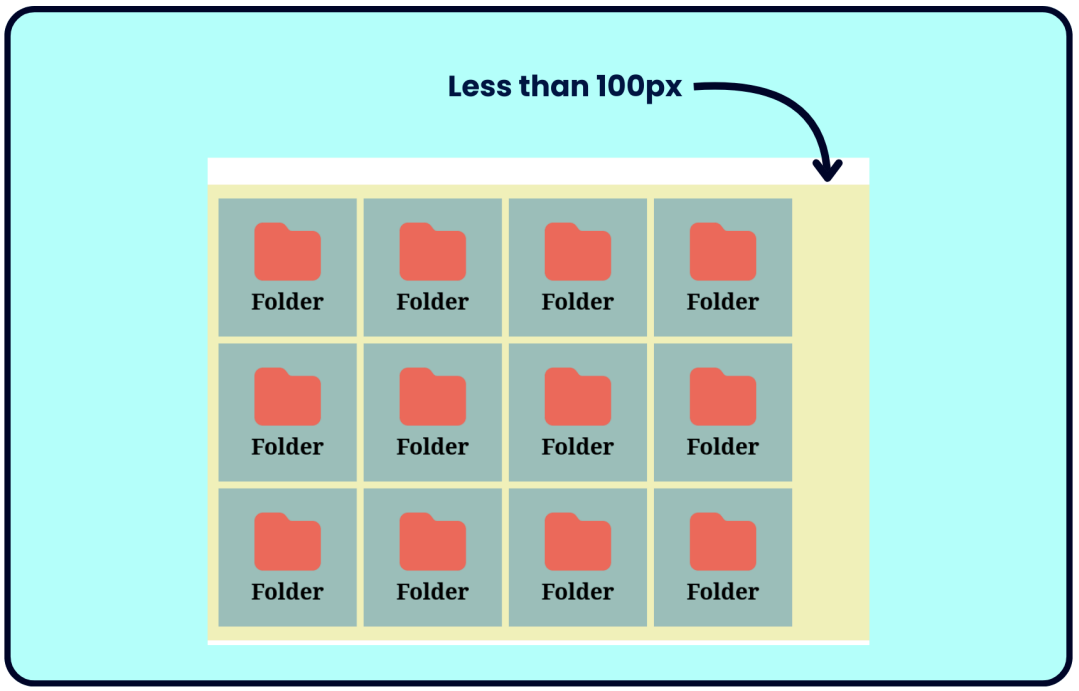
We can simply achieve it by doing this:

```
.cont{
    display: grid;
    grid-template-columns: repeat(auto-fill, 100px);
    gap: 5px;
}
```

Here, instead of defining a hardcoded number of columns, we said, Create as many as you can fit.

But there's a problem.

When the grid has some extra space available, but not enough to add another column (less than 100px), then it leaves blank space:



To handle this, we update our code to this:

```
.cont {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(100px, 1fr));
  gap: 5px;
}
```

Using the *minmax* function, all columns will be at least 100px, but when there's extra space, they'll grow equally (1fr) to fill it.

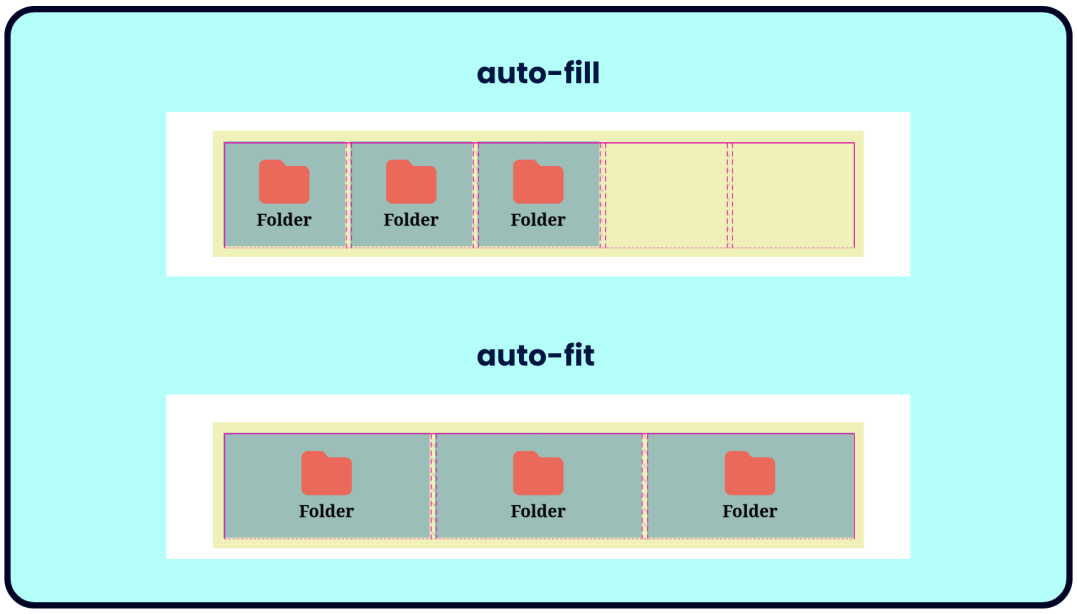
Once the space hits 100px, the columns shrink back to 100px, and a new one is added.

This creates a perfectly responsive grid.

If you want to check it out live, here is the CodePen or scan this:

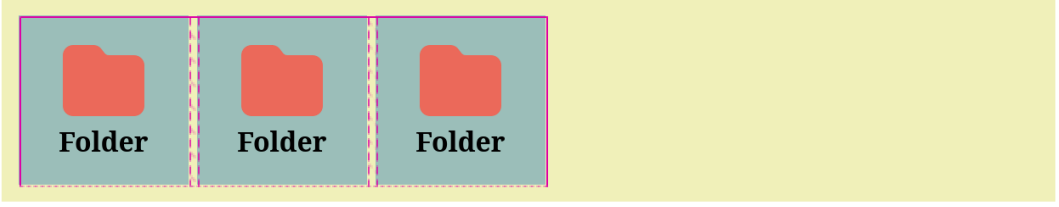
And what about *auto-fit*?

It's similar to *auto-fill*, but differs in one way. See this:



Auto-fill keeps adding columns even without elements, while *auto-fit* stops when elements are exhausted.

If we fix the columns' width to 100px, *auto-fit* would behave like this:



Here's a quick recap of what we've discussed so far:

To create a grid structure, use these properties:

1. ***grid-template-rows***: Defines the rows' size and count.
2. ***grid-template-columns***: Defines the columns' size and count.

Special values:

1. ***Fraction units (fr)***: Allocate space proportionally.
2. ***minmax()***: Set variable sizes between a minimum and maximum.
3. ***repeat()***: Repeat rows/columns easily.
4. ***auto-fit***: Creates as many tracks as can fit items.
5. ***auto-fill***: Fills the container with empty tracks after items are placed.

Combine these for a fully responsive grid with:

```
grid-template-columns: repeat(auto-fit, minmax(size, 1fr));
```

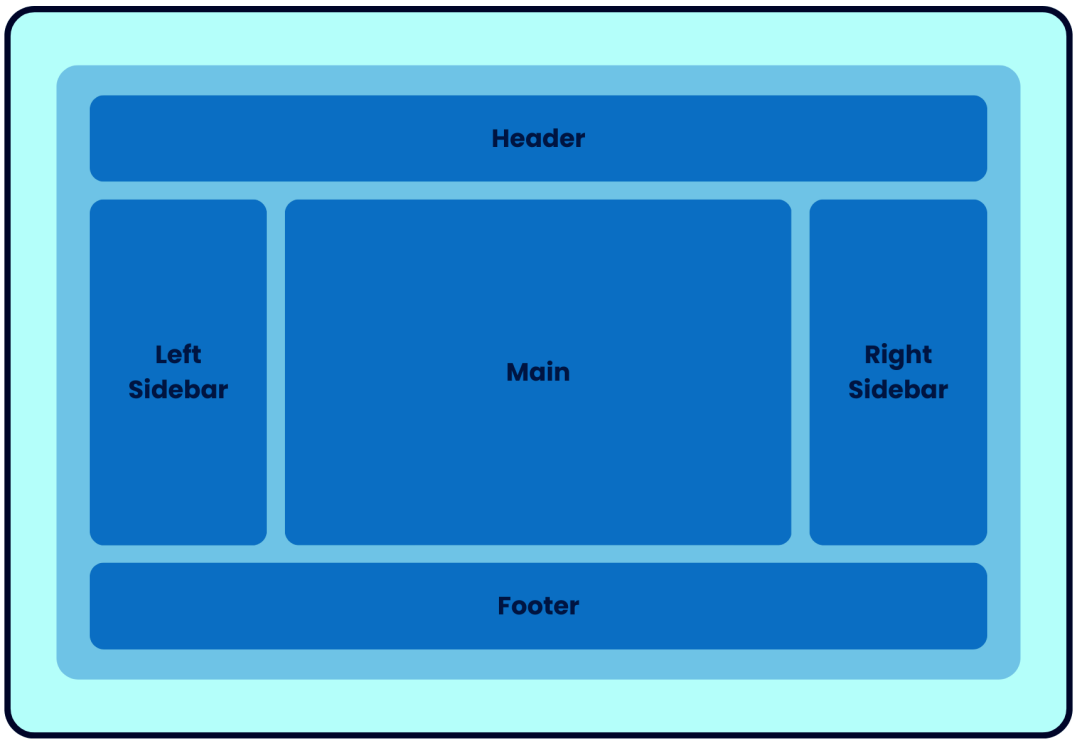
So far, we've focused on defining the grid structure but haven't specified where to place the grid items.

The next property does exactly that.

Grid-template-areas

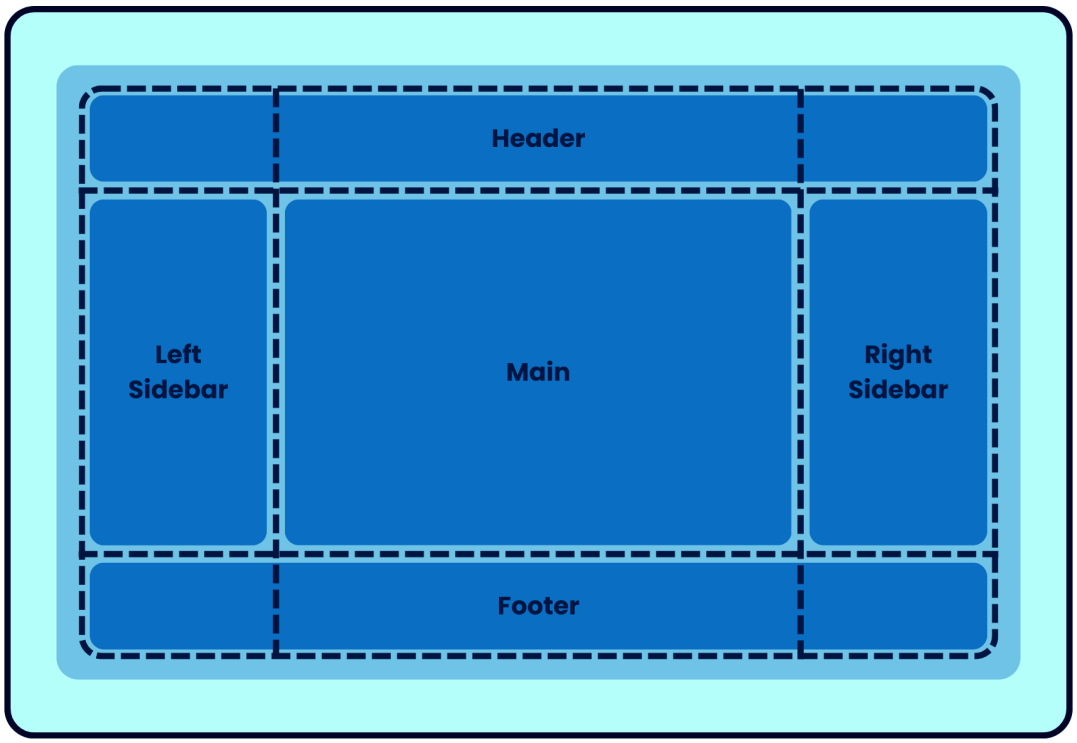
Grid-template-areas property lets you define how each grid item should be placed in the grid.

Let's understand with this holy grail layout:



What should be its grid template?

If you can't figure it out, simply draw a line on each item's edge:



We need a 3 by 3 grid like this:

```
body{
    display: grid;
    grid-template-rows: 75px 1fr 75px;
    grid-template-columns: 300px 1fr 300px;
}
```

Grid by default places one grid item in each cell. But you can change that with ***grid-template-areas***.

We have these elements:

```
<body>
    <header></header>
    <aside class="left"></aside>
    <main></main>
    <aside class="right"></aside>
```

```
    <footer></footer>
  </body>
```

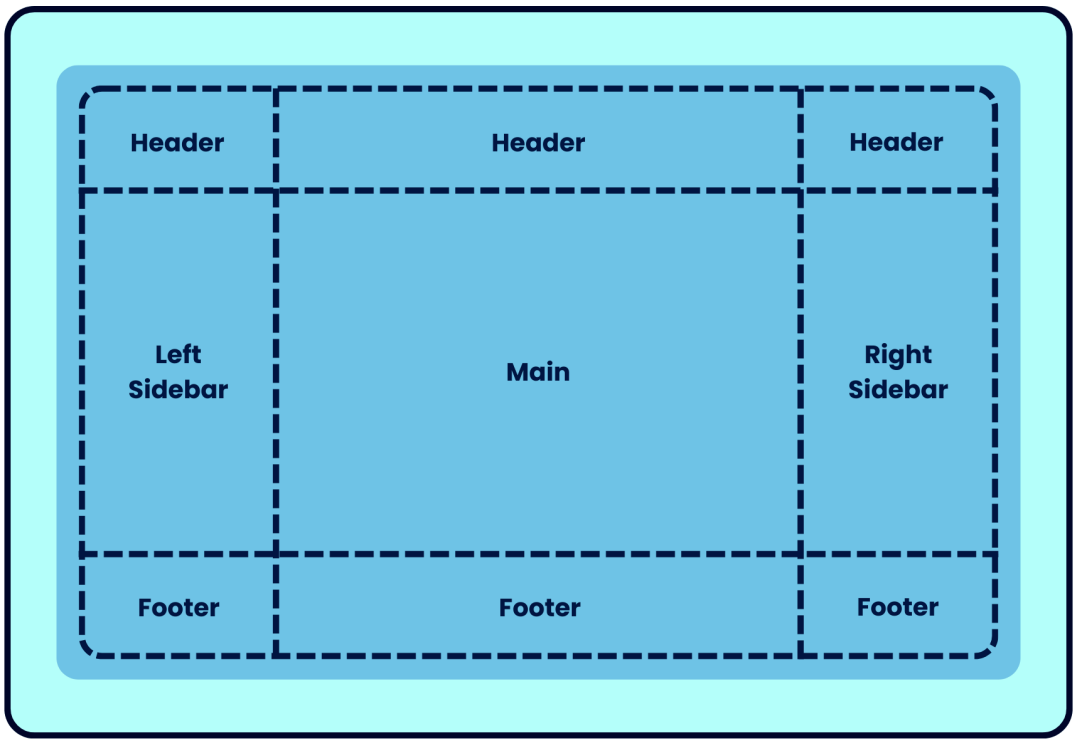
First, we need to give them all names. For that, we'll use another property called ***grid-area***, like this:

```
header{grid-area: header;}
aside.left{grid-area: l-sidebar;}
aside.right{grid-area: r-sidebar;}
main{grid-area: main;}
footer{grid-area: footer;}
```

You can give any name you want. Once done, you can use them in the ***grid-template-areas***, like this:

```
body{
    display: grid;
    grid-template-rows: 75px 1fr 75px;
    grid-template-columns: 300px 1fr 300px;
    grid-template-areas: "header header header"
                         "l-sidebar main r-sidebar"
                         "footer footer footer";
}
```

As you can see, we defined each row in inverted commas; it's as if we're filling each cell with its name. Like this:



You can also use a `.` (dot) instead of a name to leave a cell empty. Like this:

```
grid-template-areas: "header header header"
                     "l-sidebar main r-sidebar"
                     ". footer footer";
```

To check this example live, here is the CodePen link. Or scan this:

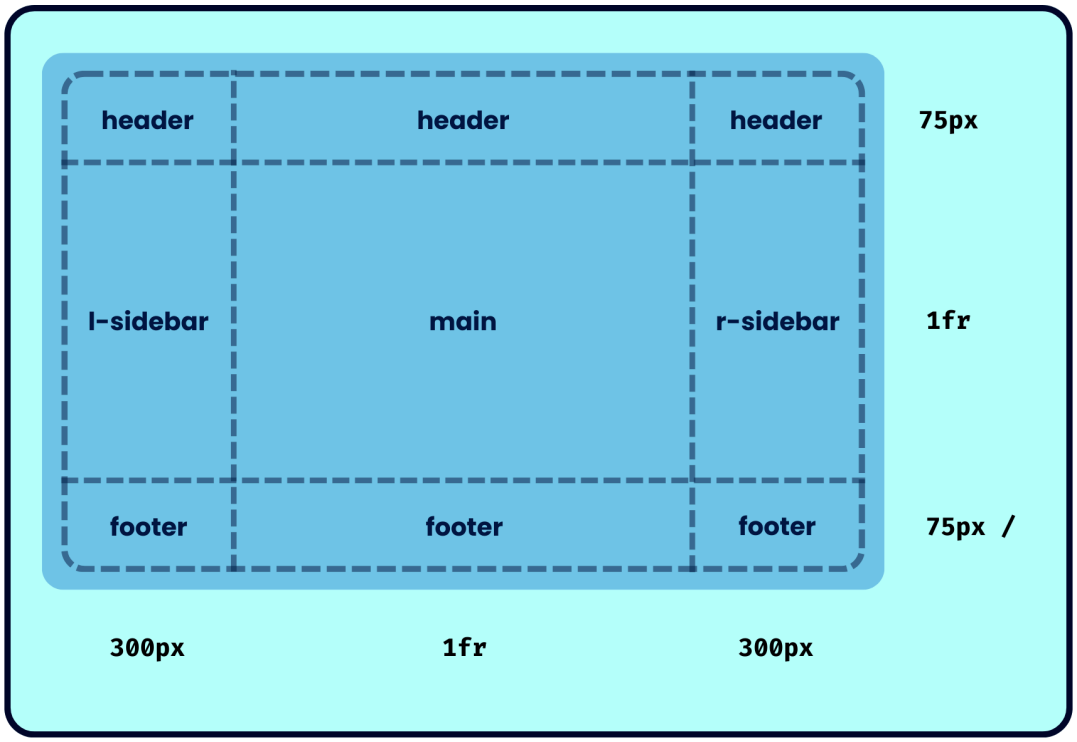
The next property takes it even to another level.

Grid-template Shorthand

We can combine ***grid-template-rows***, ***grid-template-columns***, and ***grid-template-areas*** using the ***grid-template*** shorthand. For example:

```
body{
    display: grid;
    grid-template: "header header header" 75px
                   "l-sidebar main r-sidebar" 1fr
                   "footer   footer   footer" 75px /
                    300px      1fr     300px;
}
```

So, how does this syntax work? See the layout it creates:



That's exactly how you've written it in the syntax.

Values before the / define ***grid-template-areas*** and ***grid-template-rows***, while values after it set ***grid-template-columns***.

You can also skip ***grid-template-areas***, making it ***grid-template-rows*** / ***grid-template-columns***, like this:

```
body {

    grid-template: 75px 1fr 75px / 300px 1fr 300px;

}
```

Now that we know how to create the grid structure, let's see how we can effectively place grid items in those cells.

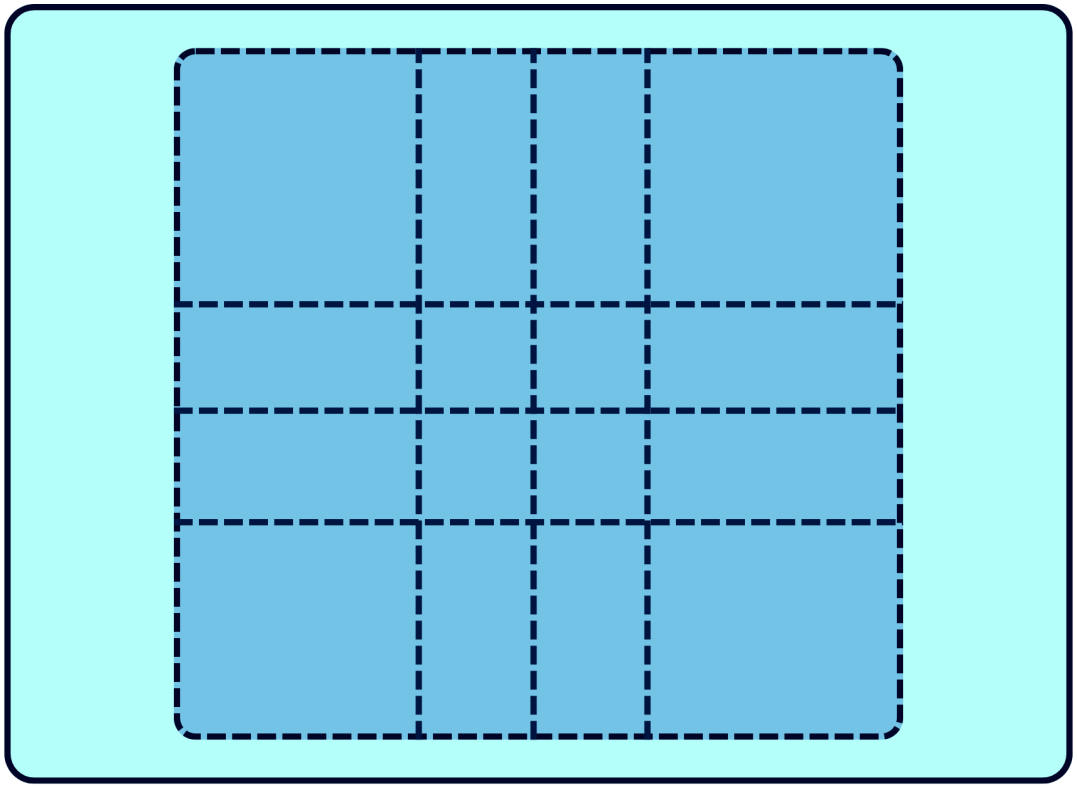
Grid Items Placement

To place grid items within a grid template, you have 4 properties:

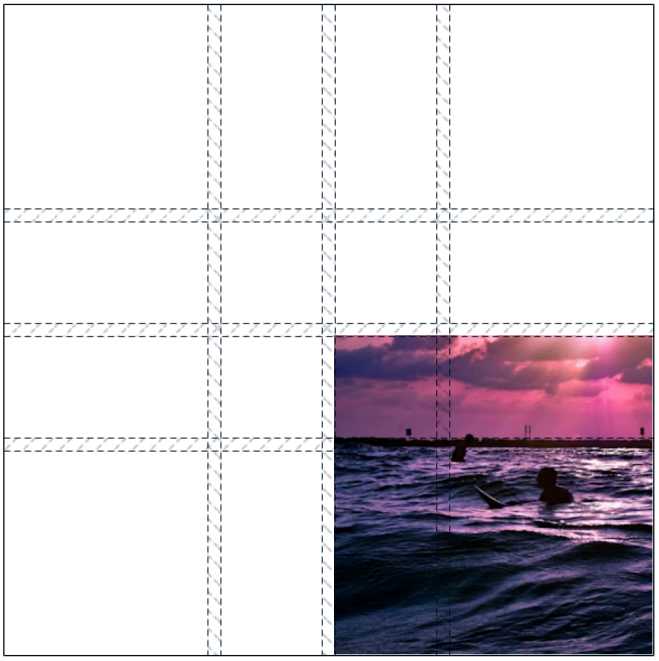
1. ***Grid-row-start***
2. ***Grid-row-end***
3. ***Grid-column-start***
4. ***Grid-column-end***

Say, we have a grid template like this:

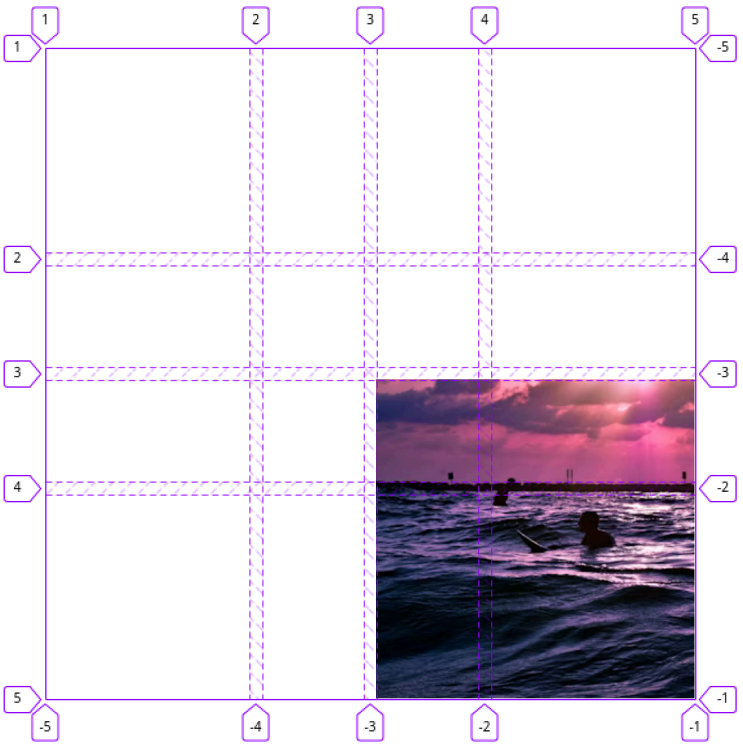
```
.cont {

    display: grid;

    grid-template: 2fr 1fr 1fr 2fr / 2fr 1fr 1fr 2fr;

}
```



How would you place an image in it like this:



First, you must know about grid line numbers. What are they? See this:



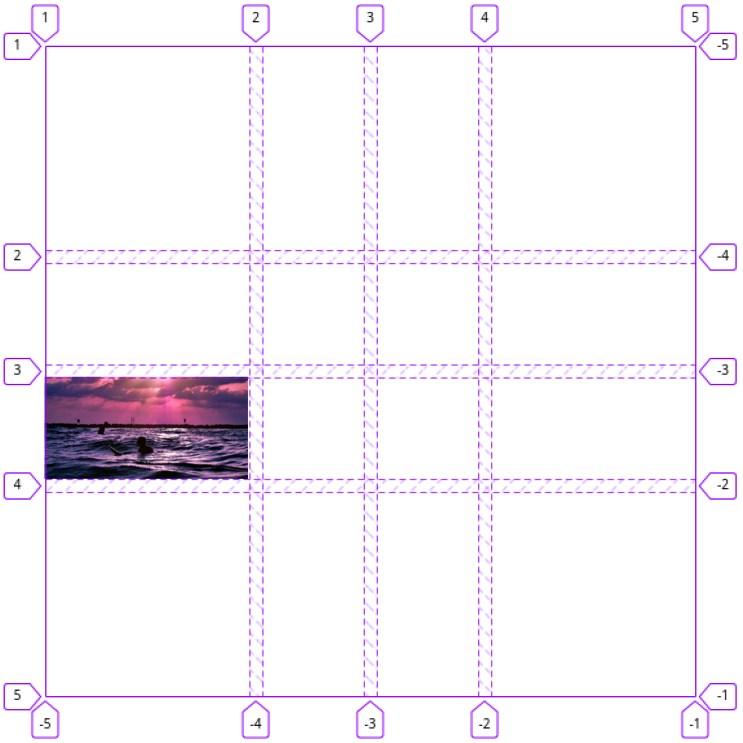
Grid lines can be referred to in two ways:

1. ***Positive***: Starts at 1 (both horizontally and vertically) and increases as you move forward.
2. ***Negative***: Starts at -1 (from the last line) and decreases as you move backward.

Grid-row-start

It defines where a grid item should start horizontally. In this case, we want it to start at the third line, so we do:

```
.img{

    grid-row-start: 3;

}
```



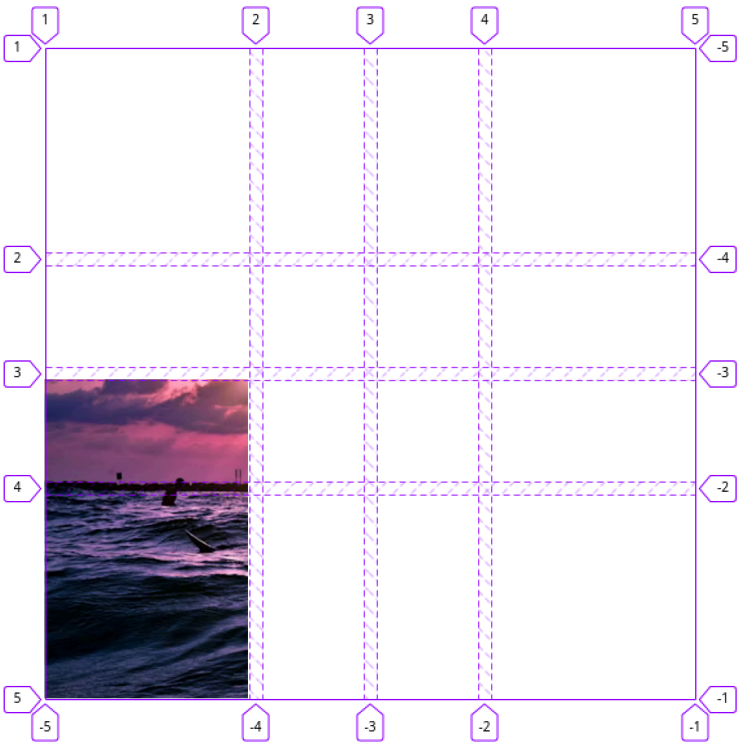
Grid-row-end

It lets you set where the grid item ends. In this case, we want to span it up to the *fifth* line. You have three options:

1. ***Positive***: Set it to 5.
2. ***Negative***: Set it to -1 (as the 5th line is the *last* line).
3. ***Span***: Set it to *span 2* (it lets you define how many tracks to span from the current one).

Like this:

```
.img{
    grid-row-start: 3;
    grid-row-end: 5;
    /* or */
    grid-row-end: -1;
    /* or */
    grid-row-end: span 2;
}
```



Note 📝

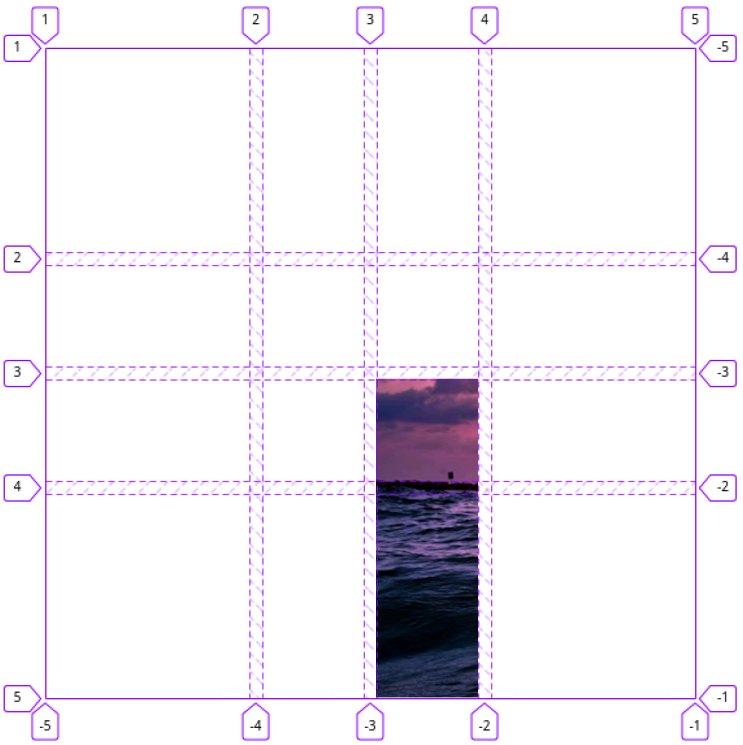
You can also use the span keyword in *grid-row-start*, but then it'd span the item from the default position.

Grid-column-start

As you'd have guessed, ***grid-column-start*** does the same thing as ***grid-row-start*** but for columns.

So, we want the image to start after the third vertical line; therefore, we'll do this:

```
.img{
    grid-row-start: 3;
    grid-row-end: -1;
    grid-column-start: 3;
}
```



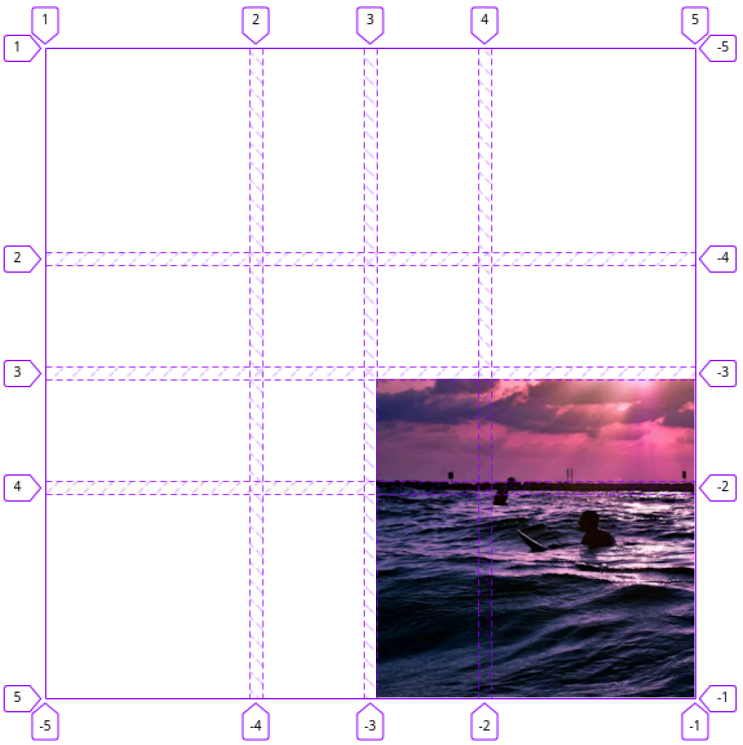
Grid-column-end

Again, it's the same as ***grid-row-end*** but for columns.

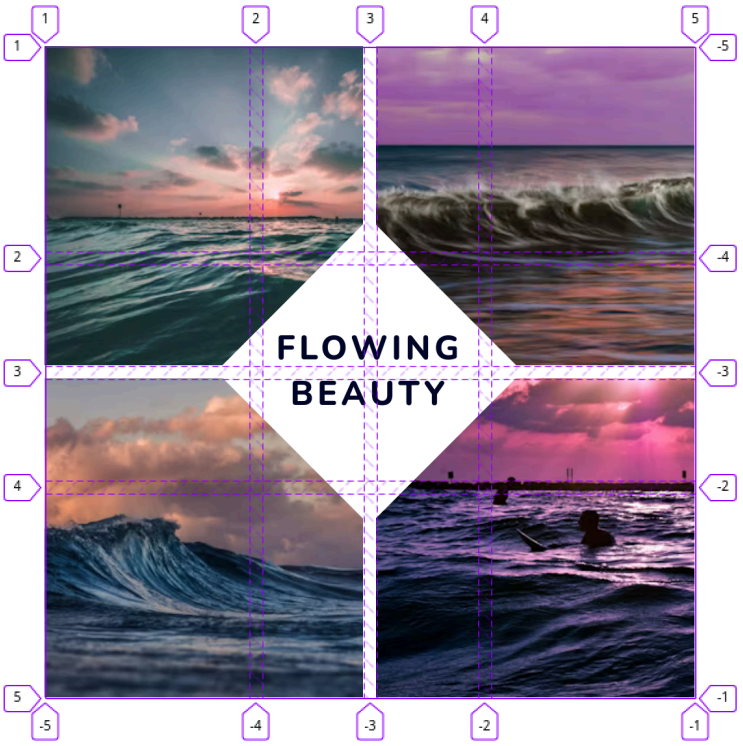
To make the image span two columns, use any one of the options similar to ***grid-row-end***:

```
.img{
    grid-row-start: 3;
    grid-row-end: -1;
    grid-column-start: 3;
    grid-column-end: -1; /* or span 2 or 5 */
}
```

And you'll get it at the right position:

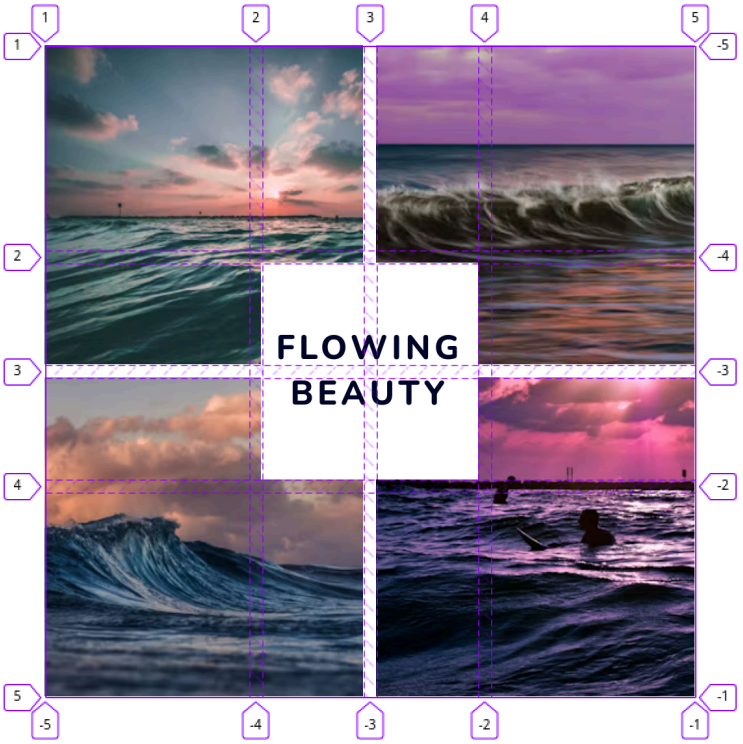


Now, from this layout, how’d you go to this:



First, place the four images in their respective rows and columns, then position the caption in the center. Like this:

```
.caption-wrapper {
    grid-row-start: 2;
    grid-row-end: 4;
    grid-column-start: 2;
    grid-column-end: 4;
}
```



Rotate the caption wrapper by 45 degrees, then rotate the text back to -45 degrees to achieve the layout.

I also used ***z-index: 2*** to position the caption above the images. Note that ***z-index*** works in flex and grid without changing the default (static) position.

Here is the live version of this example at CodePen. Or scan this:

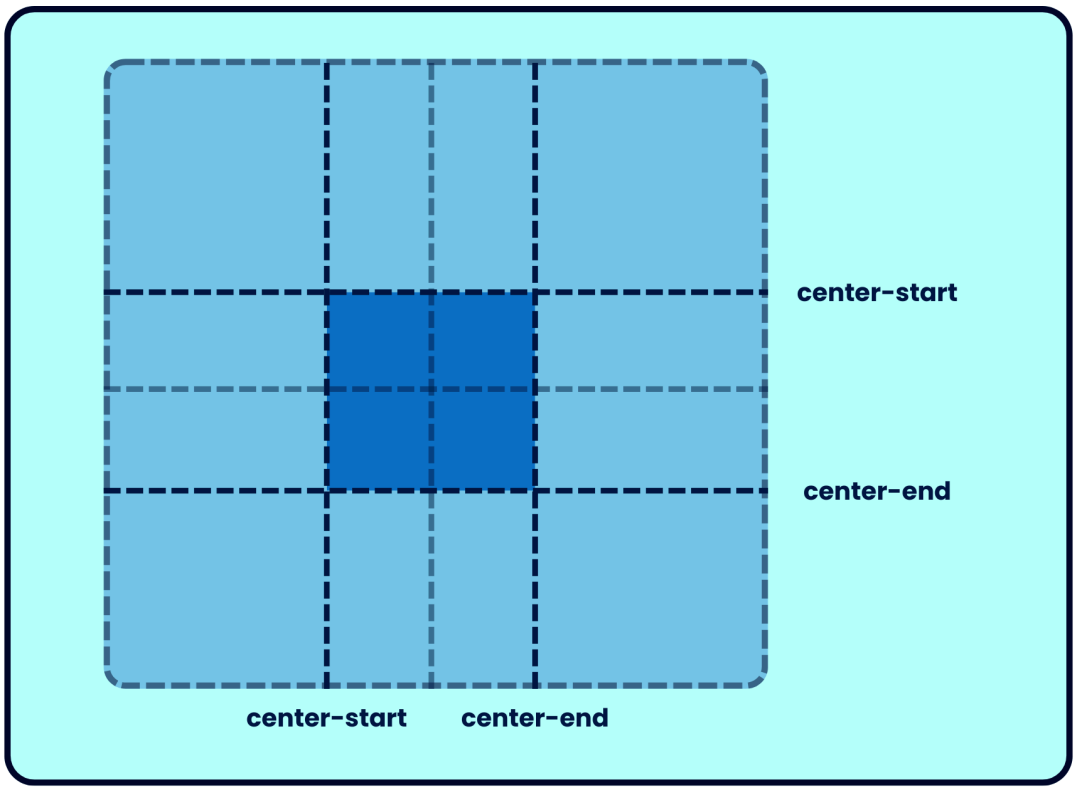
Named grid lines

You can also assign custom names to grid lines, instead of using numbers, like this:

```
.cont {
   display: grid;
   grid-template: 2fr [center-start] 1fr 1fr [center-end] 2fr
                / 2fr [center-start] 1fr 1fr [center-end] 2fr;
}
```

Simply add the line name inside the square brackets at the desired grid line's position.

It'd name the following grid lines:



Now, you can also refer to these lines with your custom names. Like this:

```
.caption-wrapper {
    grid-row-start: center-start; /* 2 */
    grid-row-end: center-end; /* 4 */
    grid-column-start: center-start; /* 2 */
    grid-column-end: center-end; /* 4 */
}
```

And do you see that blue area formed by these four lines?

You can use the common prefix **"*center*"** to refer to it. Setting this as the value for the ***grid-area*** property will place the item there:

```
.caption-wrapper{
    grid-area: center;
}
```

The only rules are to have a common prefix and **start** or **end** suffix for all four lines.

Now you know everything about grid item placement. Let's quickly revise:

1. Grid setup: Define the template, then place items using four properties:

    a. ***grid-row-start***: Start the grid line horizontally.
    b. ***grid-row-end***: End grid line horizontally.
    c. ***grid-column-start***: Start the grid line vertically.
    d. ***grid-column-end***: End grid line vertically.

2. Accepted values:

    a. ***Positive numbers***: Start at 1 and increase as you move forward.
    b. ***Negative numbers***: Start from -1 at the bottom/right, decreasing as you move backward.
    c. ***span***: Define how far to stretch from the start.
    d. ***Line names***: Custom names in ***grid-template-rows***/***columns*** or shorthand in the format *[names]*.

Grid Items Placement Shorthands

We've used these 4 properties to place the grid items:

1. ***Grid-row-start***
2. ***Grid-row-end***
3. ***Grid-column-start***
4. ***Grid-column-end***

But you can also define them with shorthands. Let's see how.

Grid-row Shorthand

The ***grid-row*** is a shorthand for ***grid-row-start*** and ***grid-row-end***.

So instead of writing this:

```
.img1 {
    grid-row-start: 1;
    grid-row-end: 3;
}
```

You can write this:

```
.img1 {
    grid-row: 1 / 3;
}
```

Just separate the values using /, where the first value is ***grid-row-start*** and the latter is ***grid-row-end***.

Grid-column Shorthand

As you'd have guessed, it's a shorthand for both columns properties:

Grid-column: grid-column-start / grid-column-end

Again, just separate the values with /, and the rest is the same.

Grid-area Shorthand

We've discussed the ***grid-area*** property twice so far:

1. To define custom area names that we can use in ***grid-template-areas***.

```
header{grid-area: header;}

Footer{grid-area: footer;}
```

2. To place a grid item in a grid area formed by four *named grid lines*.

```
.title{

    grid-area: center;

}
```

The third way is to use it as a super shorthand for the four placement properties. So, instead of doing this:

```
.img1 {

    grid-row-start: 1;

    grid-row-end: 3;

    grid-column-start: 1;

    grid-column-end: 3;

}
```

You can simply do this:

```
.img1 {
    grid-area: 1 / 1 / 3 / 3;
}
```

The values are in this order:

Grid-area: grid-row-start / grid-column-start / grid-row-end / grid-column-end

You don't necessarily have to define all 4 values; you can omit one or more, like this:

```
.img1 {
    grid-area: 1 / 1 / 3 / 3;
    grid-area: 1 / 1 / span 2; /* grid-column-end omitted */
    grid-area: 1 / 1;  /* Both end values omitted */
    grid-area: 1; /* Only grid-row-start is defined */
}
```

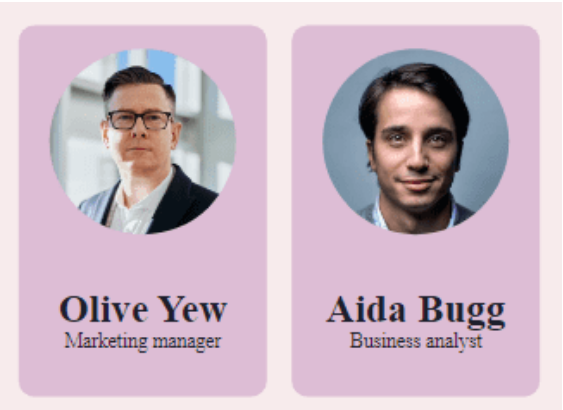
Simply put, there are three shorthands:

1. ***Grid-row***: For ***grid-row-start*** and ***grid-row-end***
2. ***Grid-column***: For ***grid-column-start*** and ***grid-column-end***
3. ***Grid-area***: For all four placement values

Next up are implicit grid tracks. What are they? Let's see.

Implicit Grid Tracks

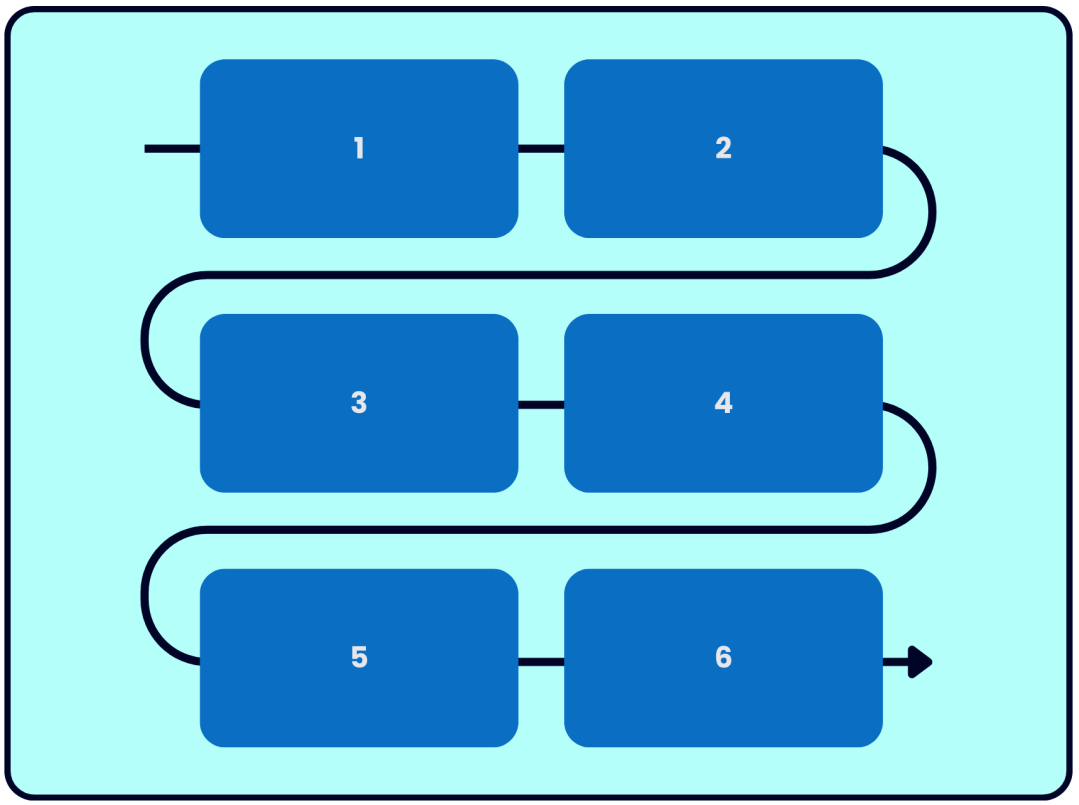
You need to create this simple layout with grid:



You can create a 1 x 2 ***grid-template: 1fr / 1fr 1fr***.

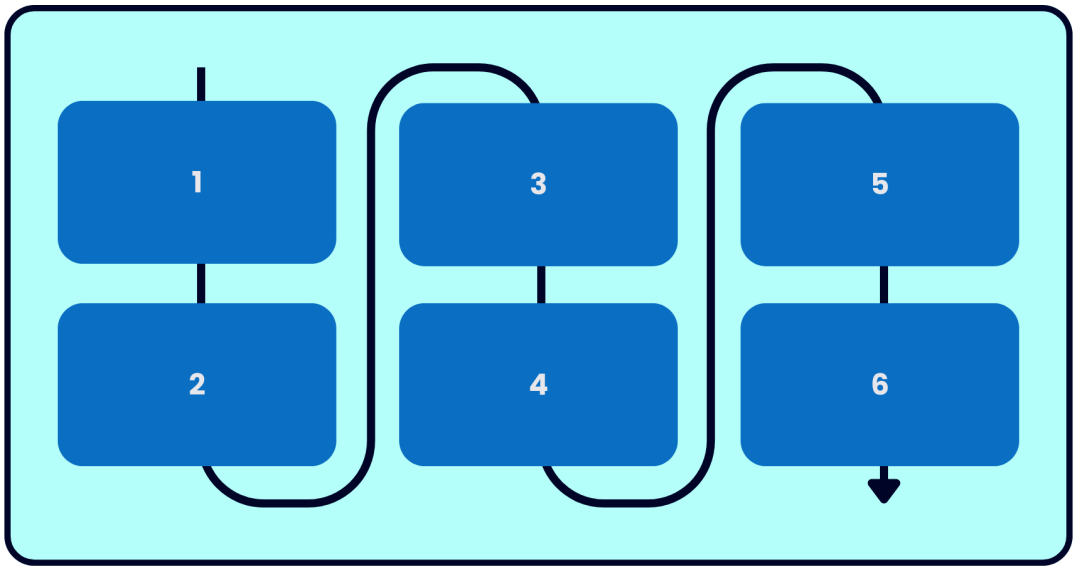
Grid-auto-flow

But what if some more cards are added to it? Well, the grid lays the items out like this:



It fills each **row** one by one. Once filled, it creates a new one. These automatically created rows are called implicit rows.

What if you want to fix the rows to two and create *implicit* columns instead? Like this:



To achieve it, you can simply do this:

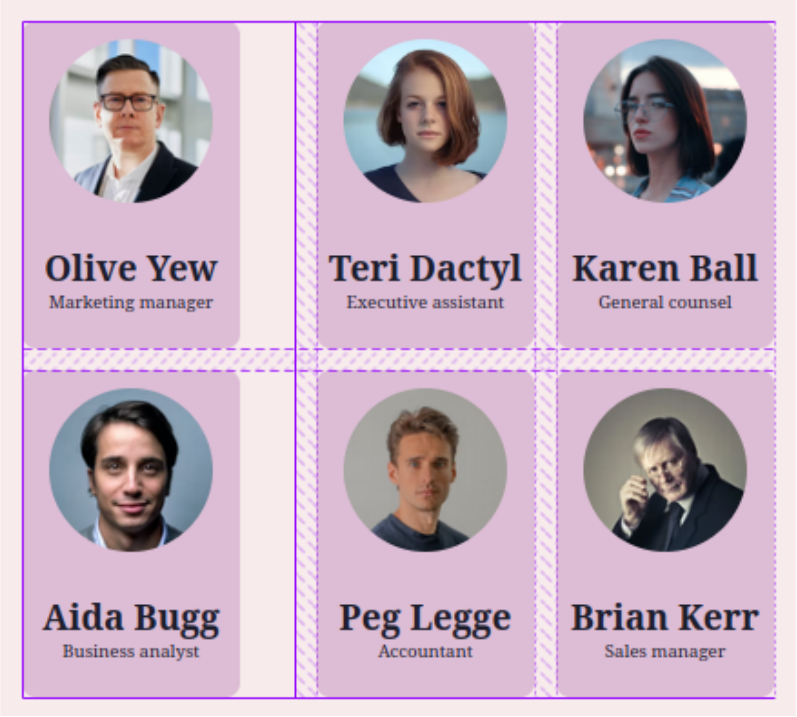
```
.wrapper{
    display: grid;
    grid-template-rows: 1fr 1fr;
    grid-auto-flow: column;
}
```

Now it'll have fixed rows but all *implicit* columns as shown above.

You can also define some *explicit* columns and the rest *implicit* like this:

```
grid-template: 1fr 1fr / 250px;
grid-auto-flow: column;
```

Then, you'll have the layout like this:



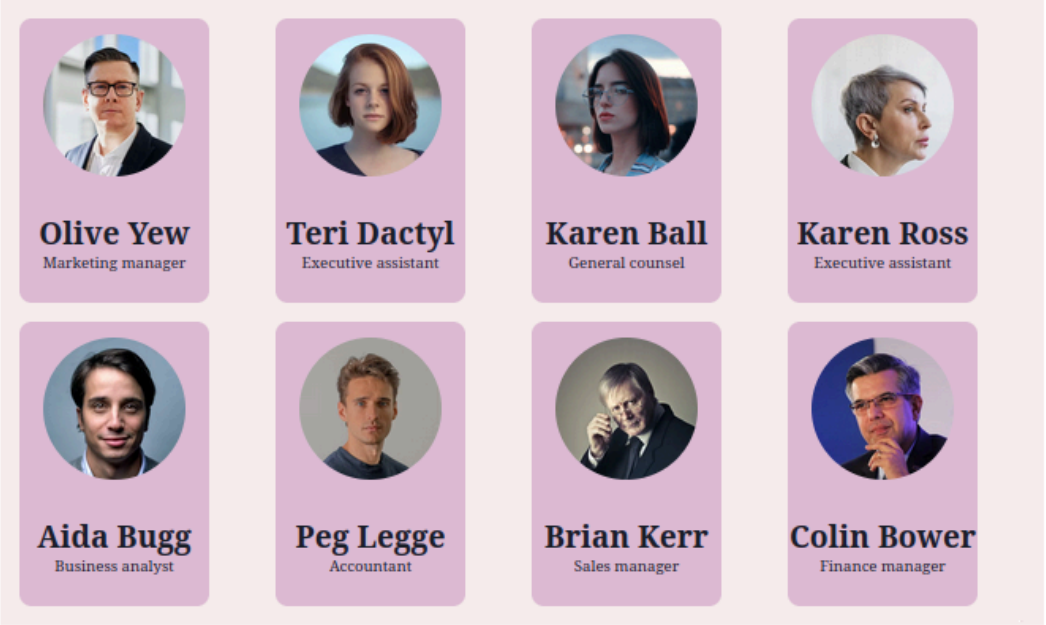
The first column is *explicit* (250px), while the other two are *implicit*. But *implicit* tracks are sized auto (equal to their content).

Grid-auto-columns

To set a width for *implicit* columns, you can use the ***grid-auto-columns*** property. Like this:

```
grid-template: 1fr 1fr / 250px;

grid-auto-flow: column;

grid-auto-columns: 250px;
```

Now, no matter how many cards you put in, it'd always put them in new columns of the size 250px:



If you want to check it live, here is the CodePen link. Or scan this:

Grid-auto-rows

As we discussed, the grid creates *implicit* **rows** by default. So, the ***grid-auto-rows*** property lets you set a size for them. Like this:

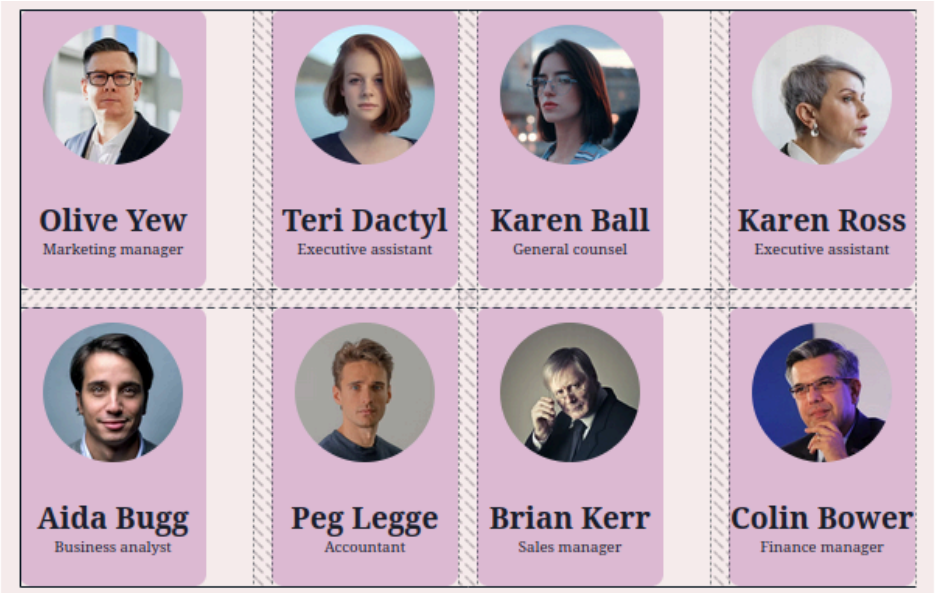
```
grid-template-columns: 1fr 1fr;

grid-auto-flow: row; /* You can omit it, default value */

grid-auto-rows: 250px;
```

> **Note** 📝
>
> For both *grid-auto-rows* and *grid-auto-columns*, you can use multiple values to create track sizes in a repeating pattern.

So, you can do it like this:

```
grid-auto-columns: 200px 1fr;
```

This will create the columns like this;



The ***grid-auto-flow*** property also accepts a third interesting value. Let's see

Dense Packing Algorithm

Other than *row* or *column*, you can also set ***grid-auto-flow*** to *“dense”*. But what it does? Let’s see an example.

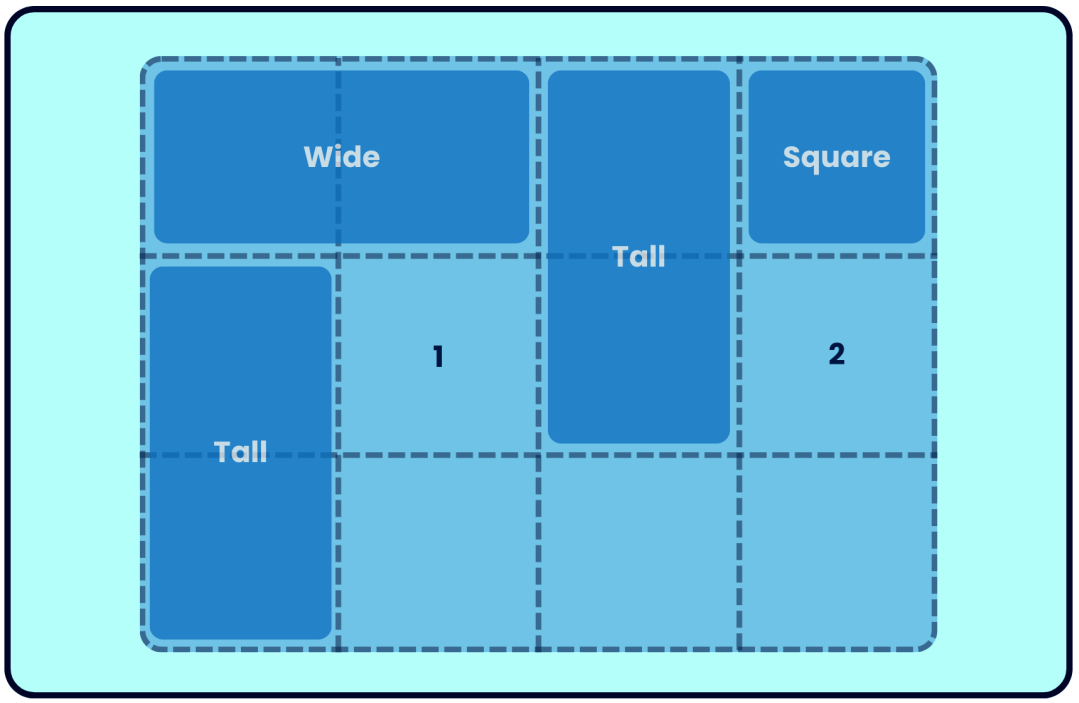
You need an image gallery with tall, wide, and square images placed randomly. Like this:

```
<div class="gallery">
    <div class="images wide"></div>
    <div class="images tall"></div>
    <div class="images square"></div>
    <div class="images tall"></div>
</div>
```

We create a squared cell grid, spanning wide images with two columns and tall ones with two rows:

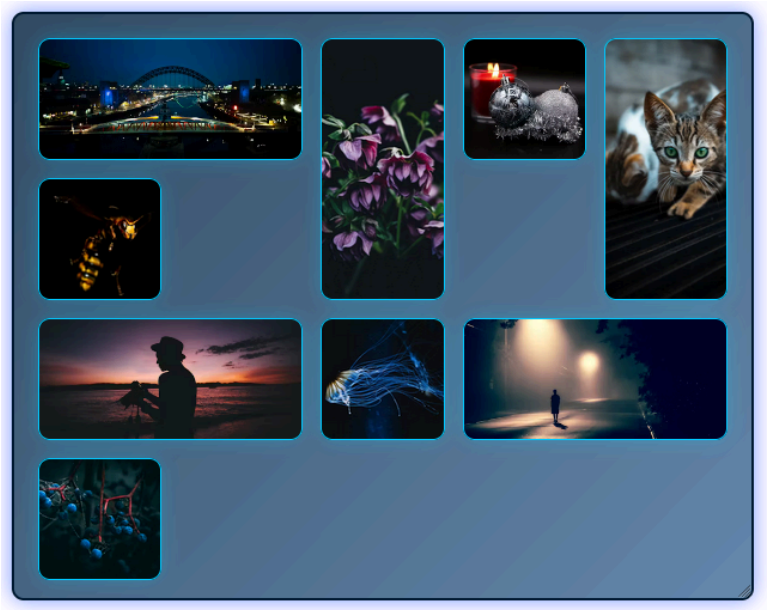
```
.gallery {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(100px, 1fr));
}
.wide {grid-column: span 2;}
.tall {grid-row: span 2;}
```

Like this:



What if you add another wide image to it?

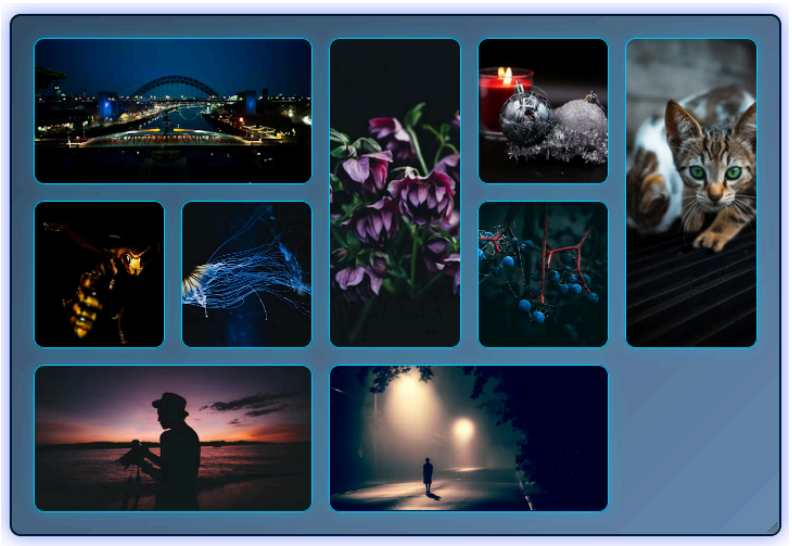
It first tries cell numbered 1, but the next cell isn’t empty. It checks cell numbered two; it’s the last cell. Thus, it’d place it in the third row. Like this:



Although the image didn't fit perfectly in those cells, some subsequent squared images could.

They’re not placed there to preserve the order in the markup.

But setting ***grid-auto-flow: dense*** for the gallery will densely pack items, even if it means changing the order. Like this:



If you want to check out its live demo, here is the CodePen link. Or scan this:

Quick recap:

When the grid runs out of space, it creates *implicit* tracks.

The ***grid-auto-flow*** property lets you choose whether they should be rows or columns.

The third value, "*dense,*" fills items tightly, ignoring their order in the markup.

To size *implicit* tracks, use: ***grid-auto-rows*** for *rows* and ***grid-auto-columns*** for *columns.*

The Grid Shorthand

This is a super shorthand for the grid, allowing you to declare all these 6 properties at once:

1. ***grid-template-rows***
2. ***grid-template-columns***
3. ***grid-template-areas***
4. ***grid-auto-flow***
5. ***grid-auto-rows***
6. ***grid-auto-columns***

It accepts values in a format similar to the ***grid-template*** shorthand but also includes *implicit* track properties.

Thus, these two pieces of code will have the same effect:

```
.cont {
    display: grid;
    grid-template-rows: 100px 2fr 100px;
    grid-template-areas: "nav nav"
                         "main main"
                         "footer footer";
    grid-auto-flow: column;
    grid-auto-columns: 300px;
}
```

And this:

```
.cont {
    grid: "nav nav" 100px
          "main main" 2fr
          "footer footer" 100px /
          auto-flow 300px;
}
```

The difference is that you can add the ***auto-flow*** keyword on one side of the / to set ***grid-auto-flow*** to that side (here, column) and optionally define the size of *implicit* tracks after that.

However, you can't define both *implicit* and *explicit* tracks on the same side. For example, this is invalid:

```
.cont {
    grid: "nav nav" 100px
          "main main" 2fr
          "footer footer" 100px /
          1fr 1fr auto-flow 300px;
}
```

Note 📝

Similar to *grid-template* shorthand, you can't use the *repeat()* function in this shorthand.

Order in Grid

Just like in Flexbox, we can change the visual order of items in a grid, regardless of their order in the markup.

Just to remind you, if you give the ***order*** to grid items like this:

```
.item1{
    order: 3;
}
.item2{
    order: 1;
}
```

The one with the lower ***order*** (item 2) will be placed first, regardless of their ***order*** in the markup.

So you give an ***order*** value to the grid items, and then it arranges them in increasing order based on that.

But if you place the item in a specific cell(s), then it'd be placed there, ignoring the order value.

Alignment in Grid

In the grid, there are 9 properties to control the alignment along the **main (horizontal) axis** and **cross (vertical) axis**. But before anything…

Reminder 🔔

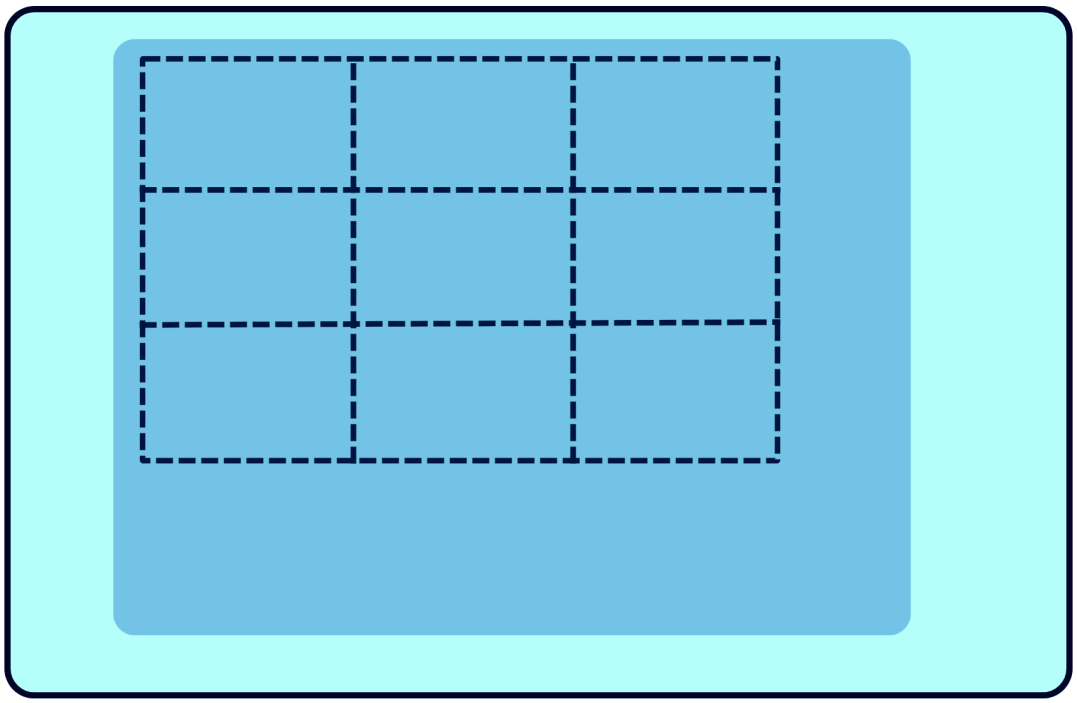
All *justify-** properties control the alignment in the *main axis*, and all the *align-** properties control the alignment in the *cross axis*.

Justify-content, align-content, and place-content

Exactly as in flex, ***justify-content*** controls the alignment of individual tracks along the *main axis*. So, if we have this:

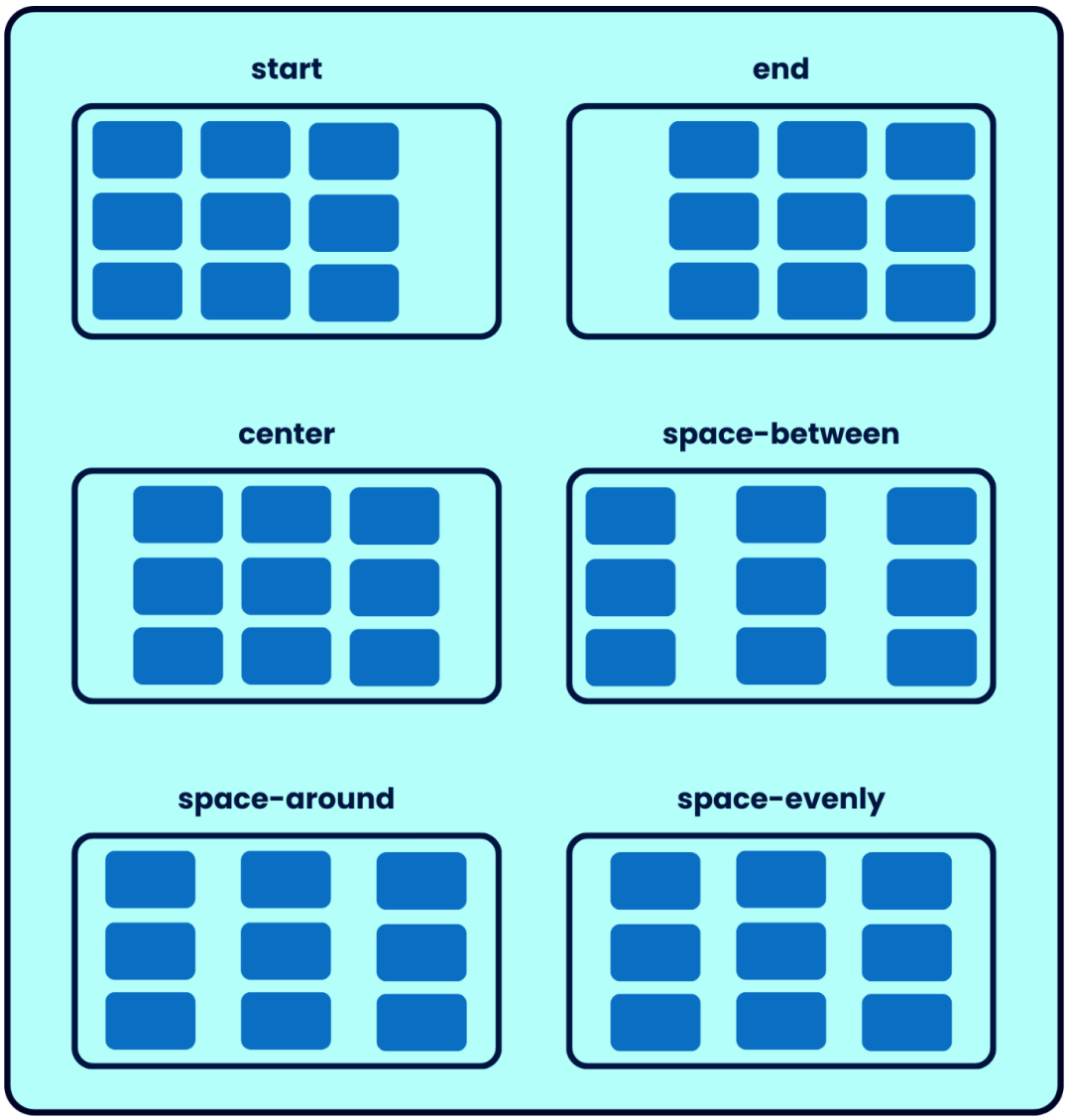
```
.wrapper{
   display: grid;
   width: 500px;
   height: 300px;
   grid-template: 100px 100px 100px / 150px 150px 150px;
}
```

We'll get this:



Columns by default are aligned to the start (left). But you can change that.

It accepts the same six values:



Similarly, ***align-content*** does the same thing but vertically (for rows).

And if you want to set both of them to the same value, use the ***place-content*** shorthand. So, instead of this:

```
justify-content: center;
align-content: center;
```

You can use this:

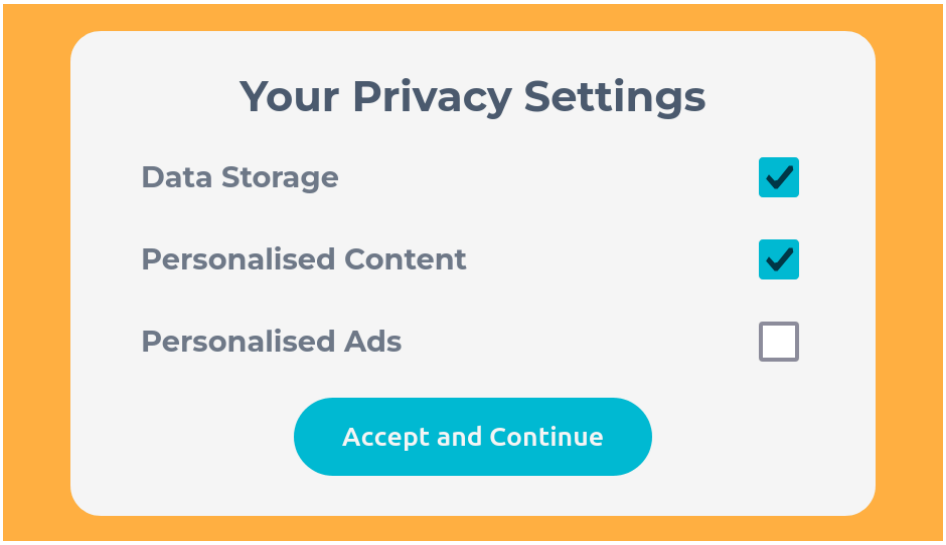
```
place-content: center;
```

Justify-items, align-items, and place-items

It lets you align the items within their tracks. It accepts these four values:

1. *Stretch (default)*
2. *Start*
3. *End*
4. *Center*

Take this layout for example:



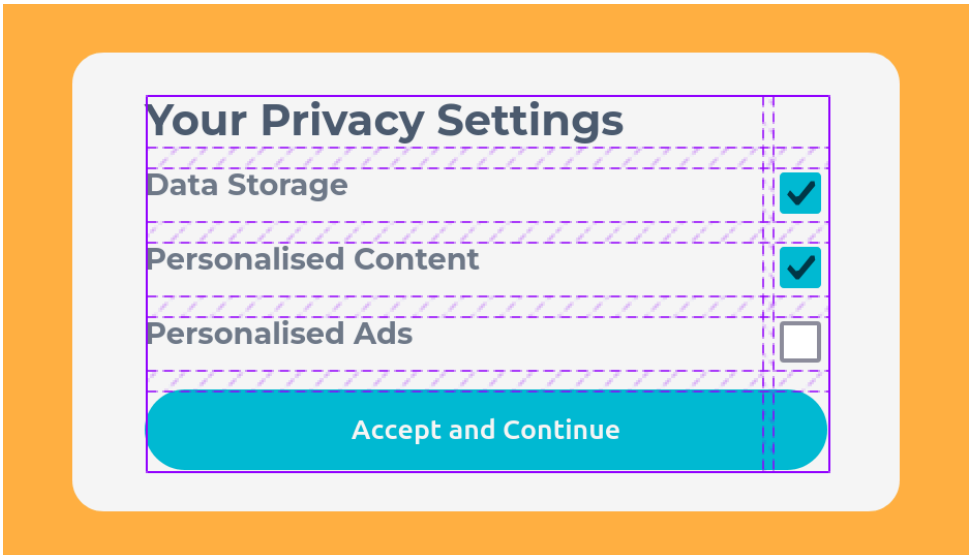
The markup is like this:

```
<form class="privacy-form">

    <h3>Your Privacy Settings</h3>

    <label for="data-storage">Data Storage</label>

    <input type="checkbox" id="data-storage">

    <!-- Other fields -->

</form>
```

We first do the basic setup like this:

```css
.privacy-form{
    display: grid;
    grid-template-columns: 1fr auto;
}
h3, button{
    grid-column: 1 / -1;
}
```

That gives us this:



We need two things:

1. Center the heading horizontally.
2. Center the labels vertically, so they align with their checkboxes.

Therefore, we add this to the container:

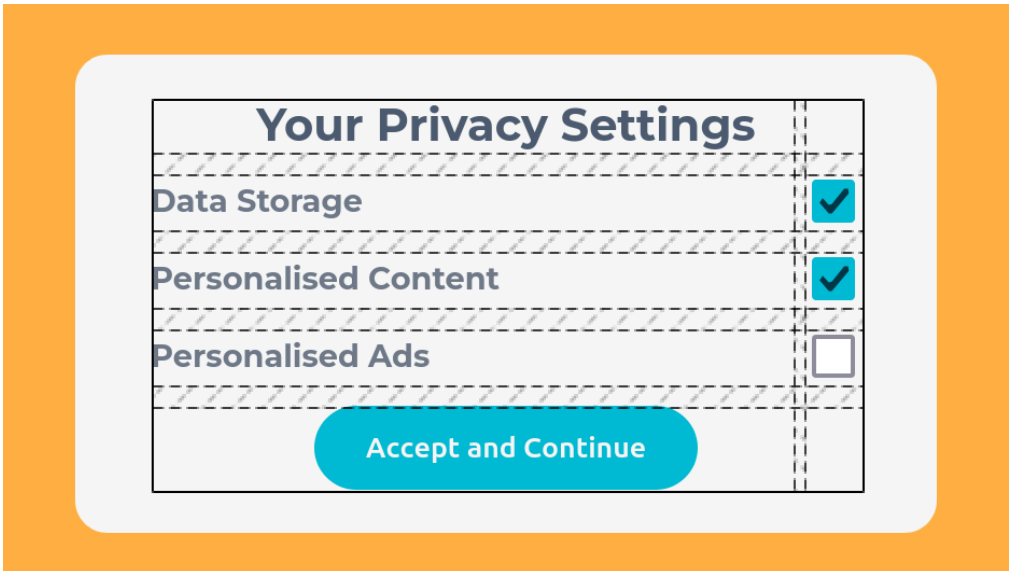
```css
.privacy-form{
    justify-items: center;
    align-items: center;
}
```

Or, as we're setting both of them to center, use the shorthand like this:

```
.privacy-form{

    place-items: center;

}
```

To avoid centering the labels horizontally, you set their width to 100% (or use the upcoming property).

Just like that, we'll get the final layout:



If you want to play around with it live, here is the CodePen. Or scan this:

Justify-self, align-self, and place-self

As you'd have guessed, these properties do the same thing as the last three, but for individual items.

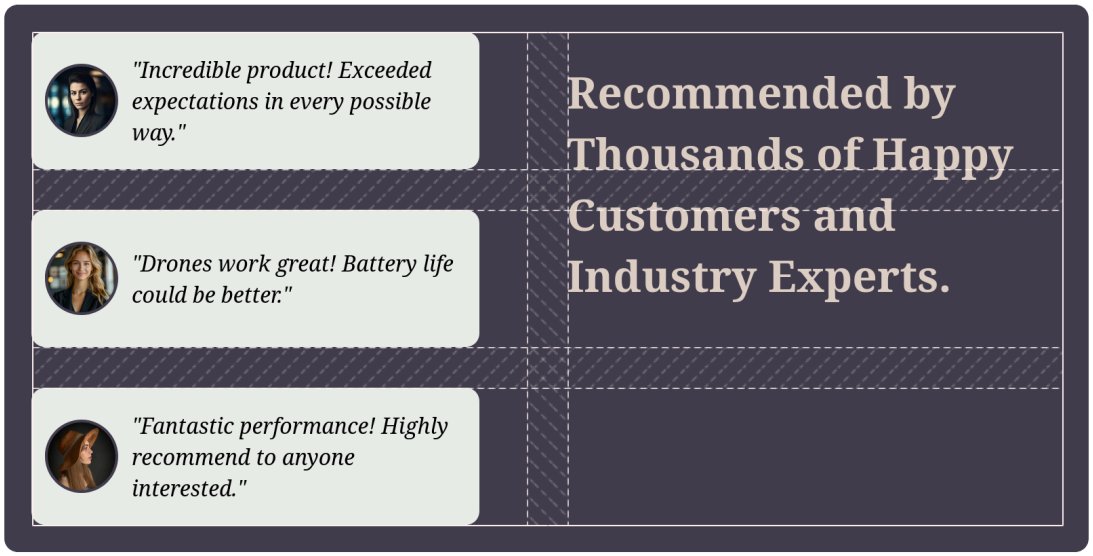
Therefore, they're applied to grid **items** and not the container.

If you want this layout:

It has 3 cards and a title So, we first create the grid-template, and place the title like this:

```
.cont{
    display: grid;
    grid-template: repeat(3, 1fr) / 1fr 1fr;
}
.caption{
    grid-row: 1 / -1;
}
```

That gives us this:



First, to prevent review cards from stretching to cover the cell, we set the width of the review cards to 85% (of the track).

Now, we need two things:

1. All cards should align to the right in the track, except for the middle one.
2. The title should be centered vertically.

For that, we add this code to the existing one:

```
.cont{
    justify-items: end; /* For all the grid items */
}
.review:nth-child(2) {
    justify-self: start; /* Change only for the second one */
}
.caption{
    align-self: center;
}
```

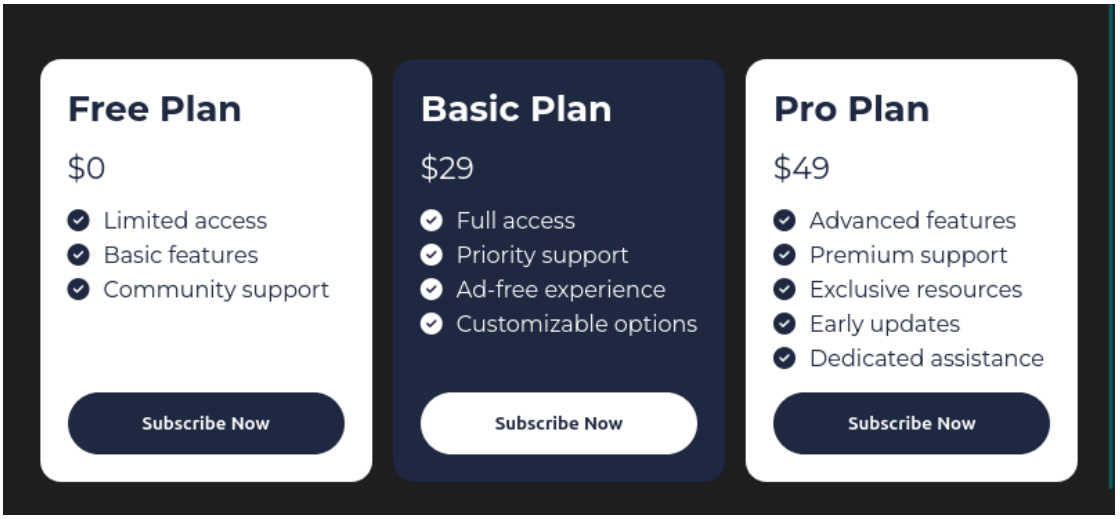
Once done, you'll get the layout. If you want to check it, here is the CodePen. Or scan this:

Here's a recap to remember all these 9 properties:

	justify-* (Main axis)	**align-*** (Cross axis)	**place-*** (Both axes)
***-content** (Aligns tracks)	justify-content	align-content	place-content
***-items** (Aligns all items)	justify-items	align-items	place-items
***-self** (Aligns one item)	justify-self	align-self	place-self

The Subgrid

Say we need three plan cards side by side, in which, even with different content lengths, all items align with each other individually:



The markup includes a main container with three cards, each containing a *plan name, price, features list*, and *button*:

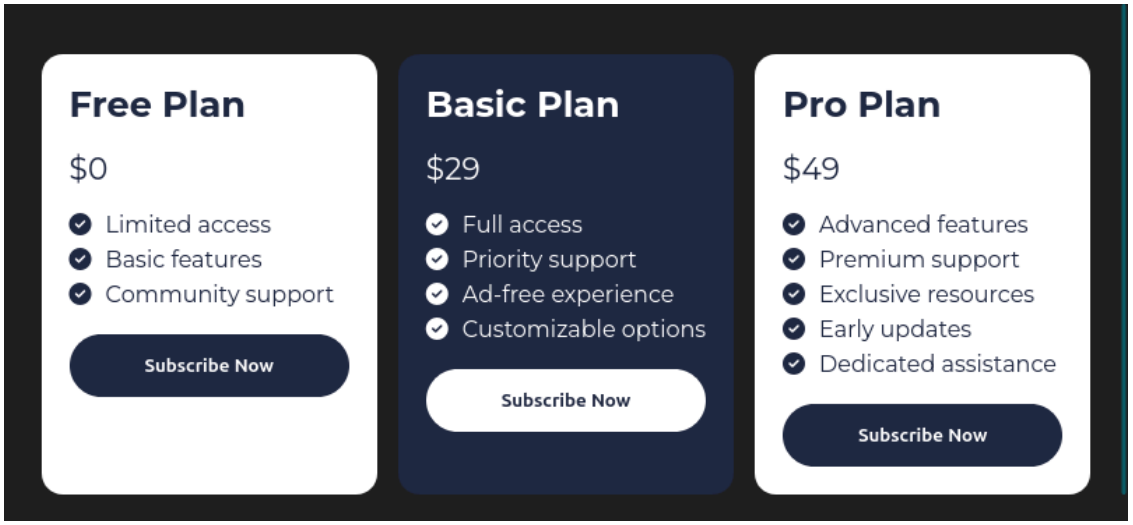
```
<div class="plans-cont">
    <div class="plan">
        <h3 class="plan-name">Free Plan</h3>
        <span class="price">$0</span>
        <div class="features">
            <!-- Varying number of features -->
        </div>
        <button>Subscribe Now</button>
    </div>
    <!-- Other two cards -->
</div>
```

For aligning the cards within the main container, we can simply do this:

```
.plans-cont {
    display: grid;
    grid-template-columns: 1fr 1fr 1fr;
}
```

The content within the cards isn't aware of each other, so they can't align.

If you try making these cards Flexbox with ***flex-direction: column***, you get this:



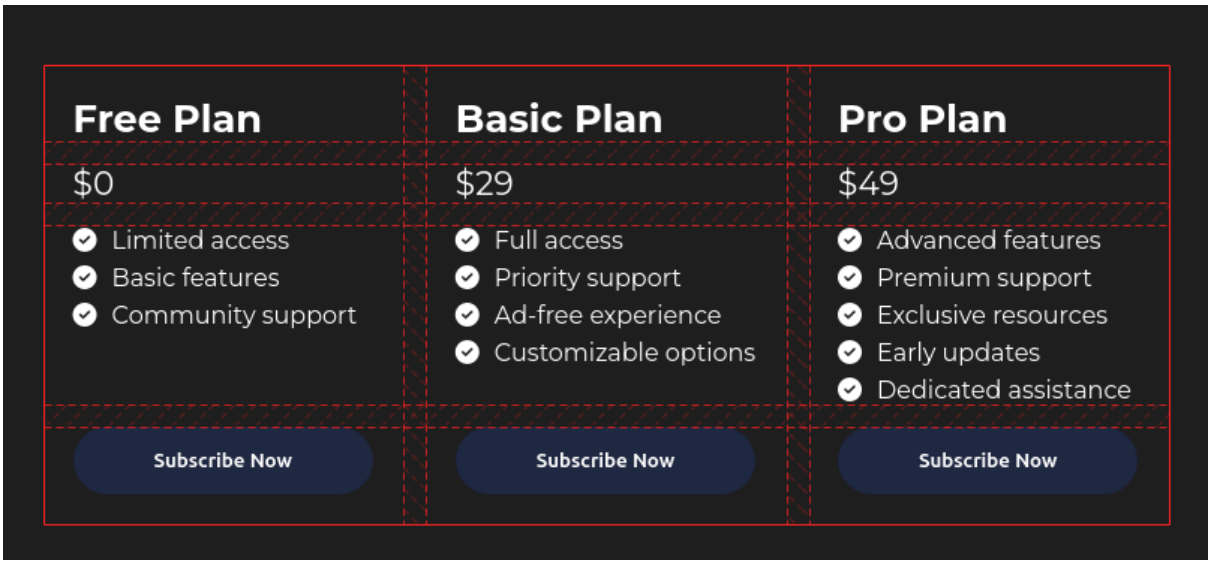
You can try adding ***justify-content***, but you'll still get misaligned content across the cards.

But if you assume for a moment that all the items are directly inside the main container and not the cards.

Then we can change the structure of that main grid like this:

```
    grid-template-columns: 1fr 1fr 1fr;
    grid-template-rows: repeat(4, auto);
```

And you'll get this:



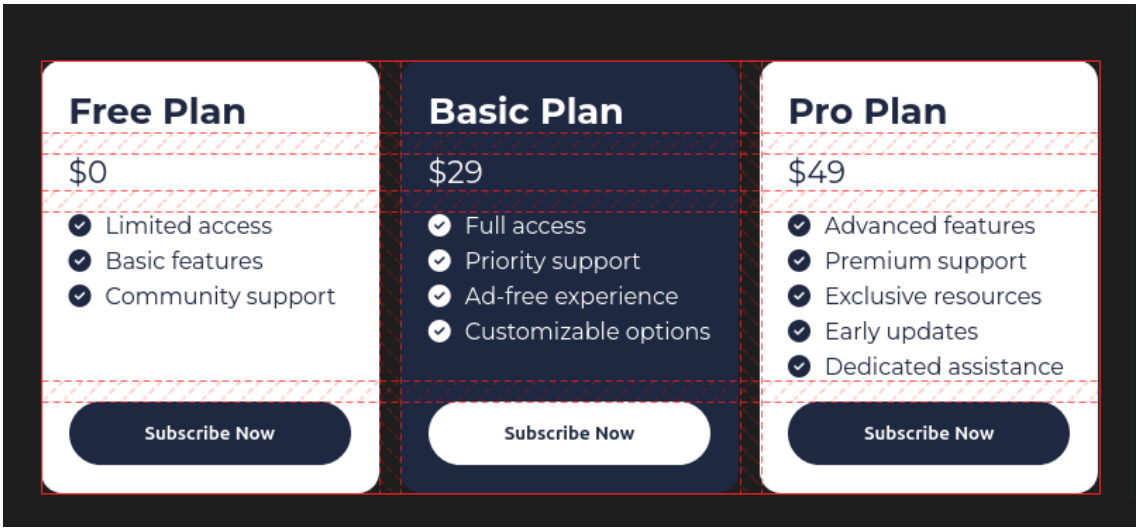
That's what subgrid do.

You don't have to remove the cards. Just span them all 4 rows and make them *subgrids* like this:

```
.plan {
    grid-row: 1 / -1;
    display: grid;
    grid-template-rows: subgrid;
}
```

It's a normal grid, but rows are set to *"subgrid"*, which is like saying inherit all the rows you're spanning of the parent grid.

Now, content of the same type is in the same row. Each row is sized based on the largest item in it, ensuring proper alignment. Like this:



Even though these items aren't direct children of the main grid, they still act as grid items of the main grid, allowing them to align with each other.

> **Note** 📝
>
> A subgrid also inherits the gap of the parent grid. But if you want, you can overwrite it in your subgrid.

If you want, you can also use subgrid in columns exactly as we did with rows, or in both.

It's a cool feature, but at the time of writing this, it's supported in 91% of the browsers. So be sure to check its compatibility.

And if you want to check this example live, here is the CodePen. Or scan this:

Flex or Grid?

How do you decide between using Flexbox and Grid?

Well, if you've made it this far, you probably have a good idea about that.

As you've seen, there's no strict rule—we've created all kinds of layouts with both. Ultimately, what matters is:

Use whichever feels easier for the situation and suits your preferences.

Here are some general tips to guide you:

1. **Dimensions:** Generally, use Flexbox for one-dimensional layouts or when items don't span multiple tracks, and grid for other cases.
2. **Structure:** In flex, the structure is determined by the items, while in the grid, we define it and items fit into it. Thus, flex, when items control the container, and grid when the container controls the items' size.
3. **Overlapping:** If you want items to overlap, most probably, you'll need a grid.
4. **Both:** If you've noticed in many of our layouts, we used both of them. So, why not utilize both of their powers?

Again, these are general ideas; you can break them. It all comes down to intuition. Which comes with practice. So…

Build layouts, experiment, break, and fix things.

The goal is to get the job done, not argue about which one is better. Both have their place.

Wrap up

Is it the end of the flexbox or grid? Definitely no!

In the programming world, everything is evolving each day. So…

To truly master them, keep exploring them, their tips, and tricks.

And if you have any doubts or suggestions, or just want to say hello, you can email me at utsav@flexicajourney.com.

And if you think this book made you a better CSS developer, or you learned something new…

Please consider recommending it to others who might find it useful.

Your support means a lot, and I can't thank you enough for helping me inspire more creativity in web design.

Share it on social media or simply celebrate—you achieved something today!

Thanks for reading, and keep rocking!

_____ *THE END* _____