CSS for beginners – 2

POSITIONING

In CSS, the **position** property controls how an element is placed on a web page. It decides **where** the element will appear and how it behaves when the page is scrolled or resized.

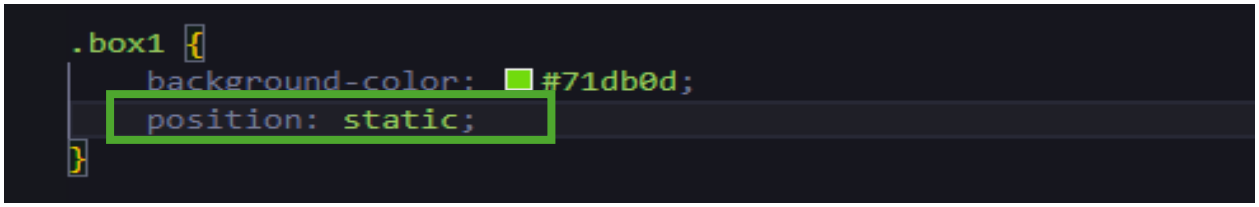
Types of Positioning in CSS

◆ **Static (Default)**

📌 **Definition:**

- Every element in HTML is **by default** position: static;.
- Elements appear **one after another, top to bottom, left to right**.
- It does **not allow custom positioning** like moving elements with top, left, right, bottom.

🔻 **Example:**



🔻 **Relatable Example:**

🏠 **"Your House Address"** – It has a **fixed, default place**, and no one moves it randomly.

◆ **Relative**

📌 **Definition:**

- Moves the element **relative to its original position** using top, bottom, left, right.
- Other elements **do not move**, only this element shifts.

🔻 **Example:**

```
.box2 {
    background-color: #14c3c9;
    position: relative;
    top: 50px;
    left: 50px;
}
```

🔻 **Relatable Example:**

🖌️ **"Cricket Pitch"** – If the umpire moves **20 steps left**, he moves **relative to his original position**, but the pitch remains unchanged.

◆ **Absolute**

📌 **Definition:**

- Removes the element from the normal document flow.
- Positions the element **relative to its nearest positioned ancestor** (not static).
- If there is **no positioned ancestor**, it will position **relative to <html> (the entire page)**.

🔻 **Example:**

```
.box3 {
    background-color: #db8719;
    position: absolute;
    top: 0px;
}
```

🔻 **Relatable Example:**

📍 **"Google Maps Location Pin"** – No matter where the map moves, the pin stays fixed **relative to the map, not the screen**.

◆ **Fixed**

📌 **Definition:**

- Positions the element **relative to the viewport (browser window)**.

- It **never moves when scrolling**.

🔻 **Example:**

```
.box4 {
    position: fixed;
}
```

🔻 **Relatable Example:**

📱 **"Instagram navigation Bar"** – No matter how much you scroll down, the **navigation bar stays at the bottom**.

◆ **Sticky**

📌 **Definition:**

- Acts like **relative** until you scroll, then it sticks to a defined position (e.g., top: 0px;).
- It **only sticks when scrolling past a certain point**.

🔻 **Example:**

```
.box1 {
    background-color: #71db0d;
    position: static;
}
```

🔻 **Relatable Example:**

🛺 **"Zomato Cart"** – As you scroll down, your cart **sticks to the side** so you don't lose track of what's added.

FLEXBOX

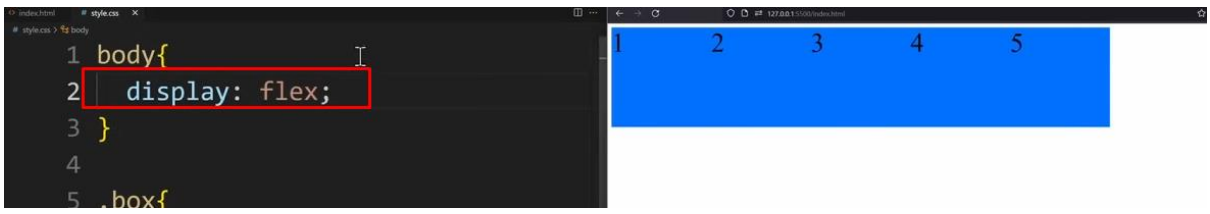
Flexbox (Flexible Box Layout) is a powerful CSS layout system designed to arrange elements efficiently in one-dimensional space (either a row or a column). It simplifies alignment, spacing, and distribution of elements within a container.

Flexbox is really important because it simplifies layout design by allowing elements to be easily aligned, spaced, and resized within a container. Unlike traditional methods like floats or positioning, Flexbox dynamically adjusts items to fit available space, making responsive design more efficient. It enables both horizontal and vertical centering, equal spacing, and flexible element distribution without complex calculations. This makes it ideal for creating navigation bars, grids, and modern UI components while reducing the need for extra CSS rules and hacks.

1. Flex Container and Flex Items

- The **flex container** is the parent element that holds **flex items**.
- Flex items are the direct children of the flex container.

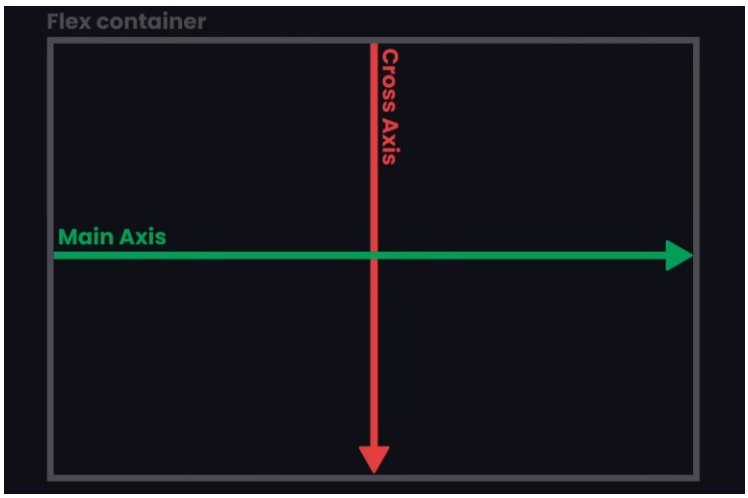
To enable flexbox, you apply display: flex; to a container, making all its children flex items.



2. Main and Cross Axis

- The **main axis** is the primary direction in which flex items are laid out (horizontal by default).
- The **cross axis** is perpendicular to the main axis.

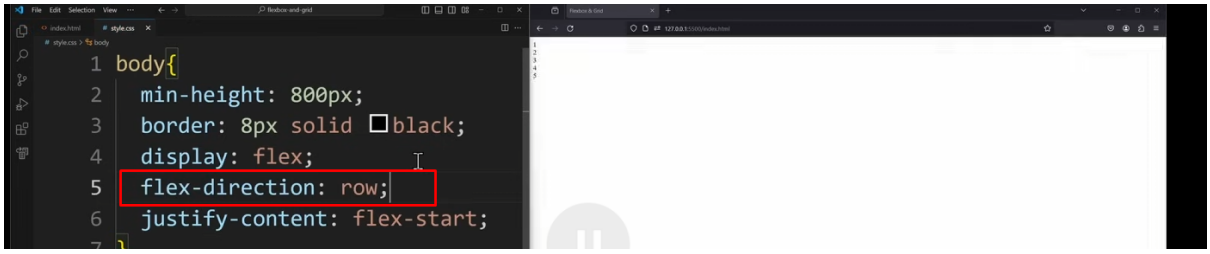
By changing flex-direction, you can switch between row-based and column-based layouts.



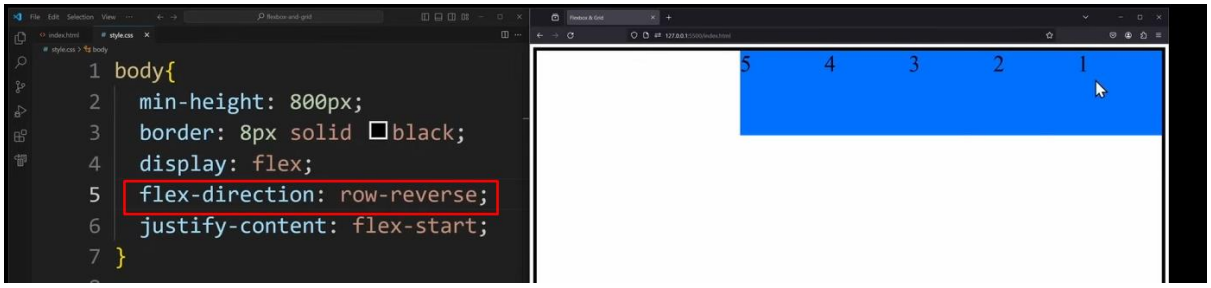
3. Key Flex Properties

a) **flex**-direction (Defines Layout **Direction)**

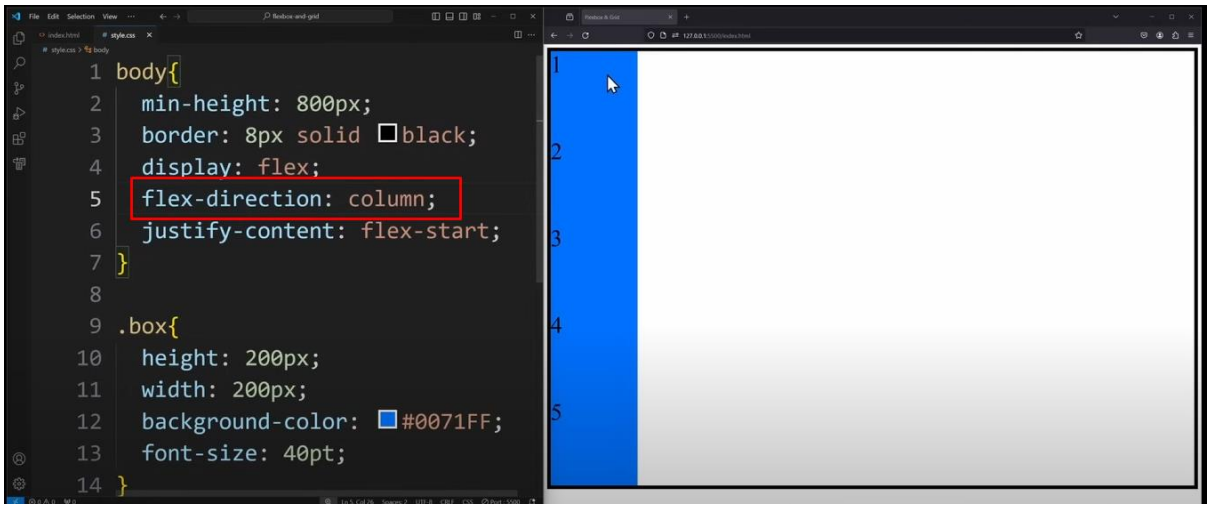
- **row** (default) – Items are placed left to right.



- **row-reverse** – Items are placed right to left.



- **column** – Items are placed top to bottom.

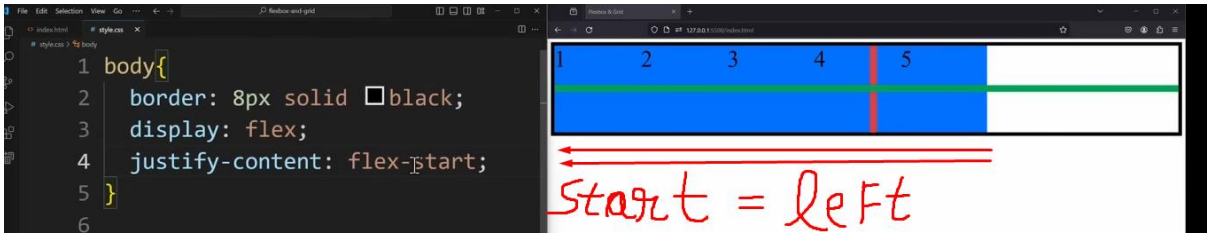


- **column-reverse** – Items are placed bottom to top.

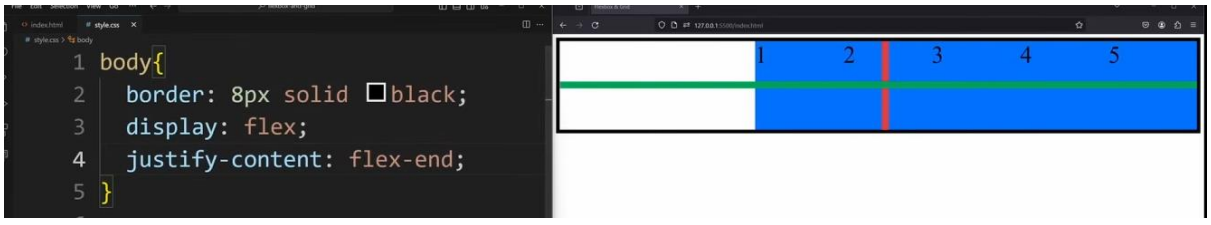
b) justify-content (Aligns Items along the Main Axis)

Controls how space is distributed between flex items along the main axis.

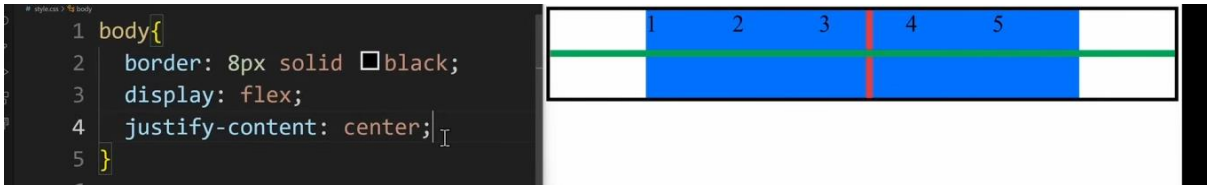
- **flex-start** – Items align at the start.



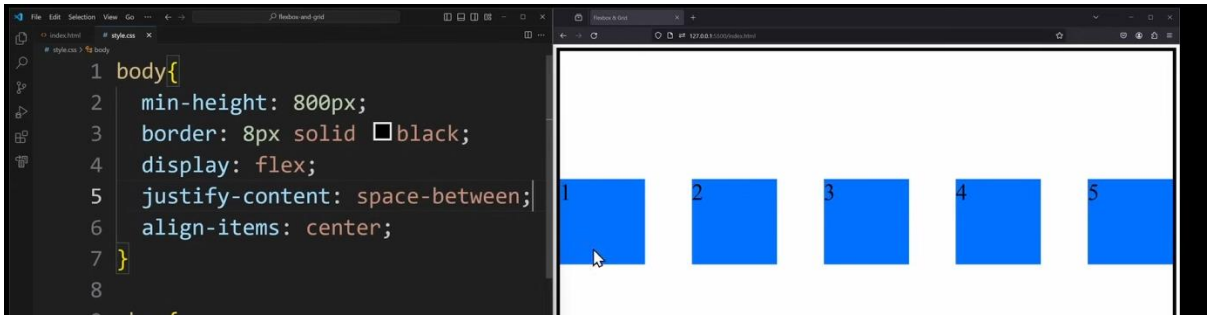
- **flex-end** – Items align at the end.



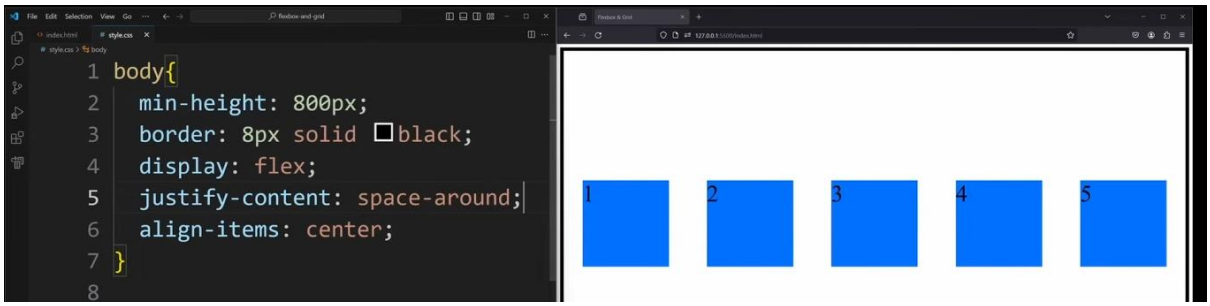
- **center** – Items are centered.



- **space-between** – Items are evenly spaced, first at start, last at end.

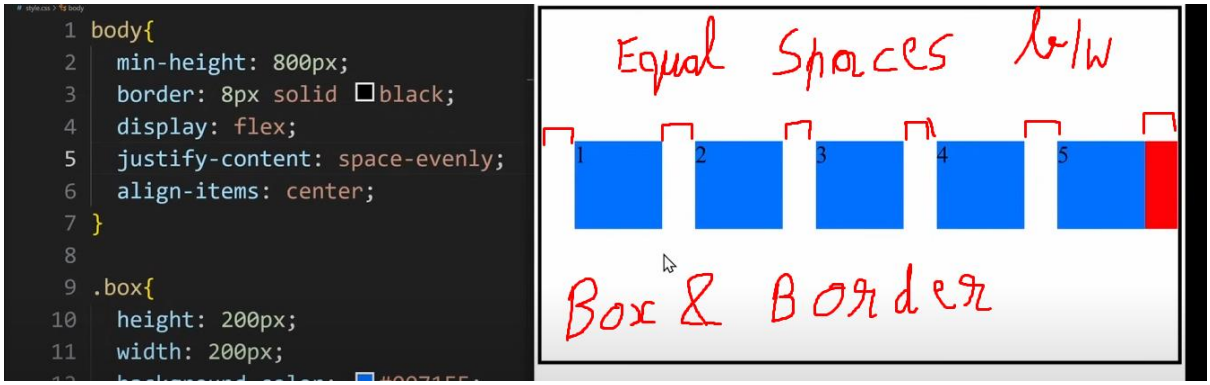


- **space-around** – Items have equal space around them.



You may notice that the space between the very edges near the border of the window is half that of the space between the boxes so to fix that we can use

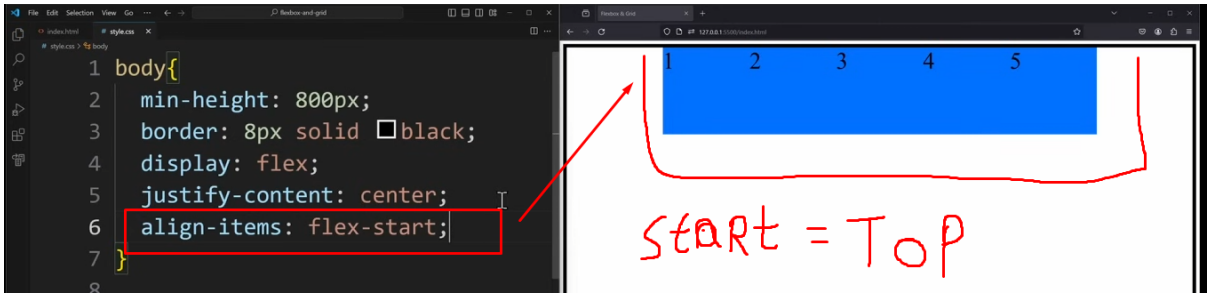
- **space-evenly**



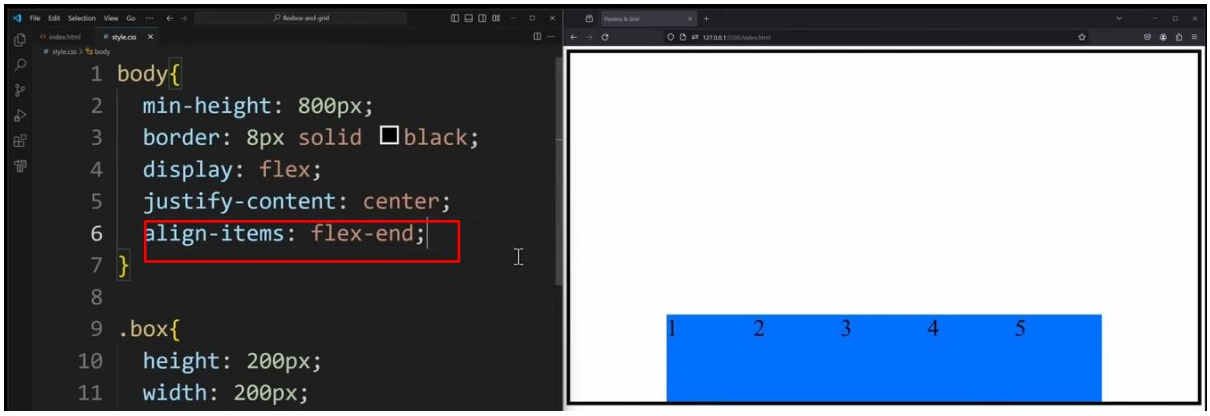
c) align-items (Aligns Items Along the Cross Axis)

Controls how items align in the perpendicular direction.

- **flex-start** – Items align at the start of the cross axis.



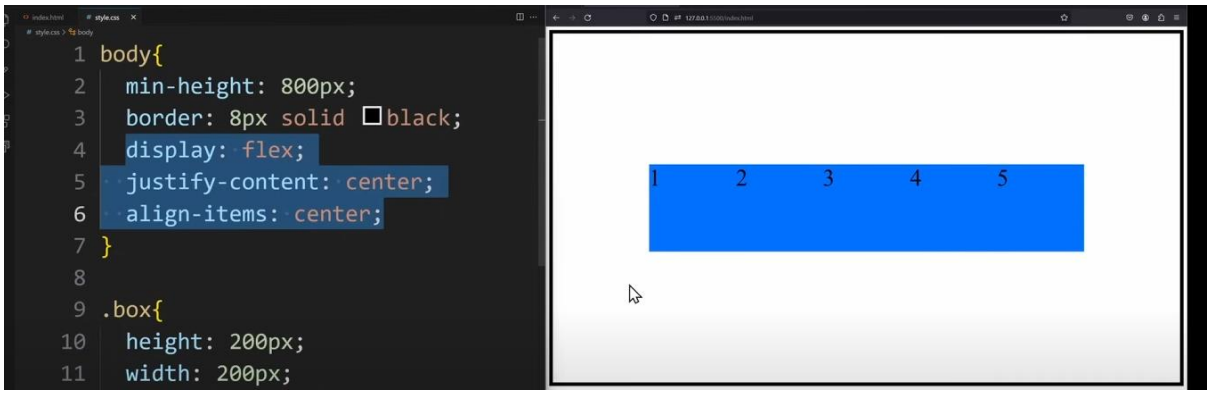
- **flex-end** – Items align at the end of the cross axis.



Very important- HOW TO CENTER A DIV?

Set align-items property to :

center – Items are centered.

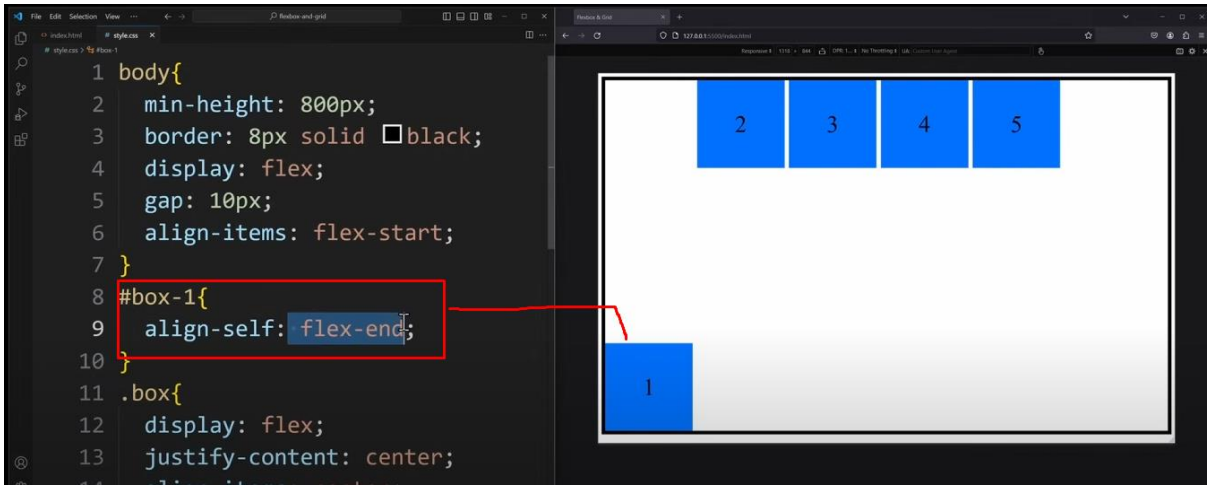


<u>Some other item alignments</u>

- **stretch** – Items stretch to fill the container.
- **baseline** – Items align based on their text baseline.

d) align-self (Overrides Alignment for a Single Item)

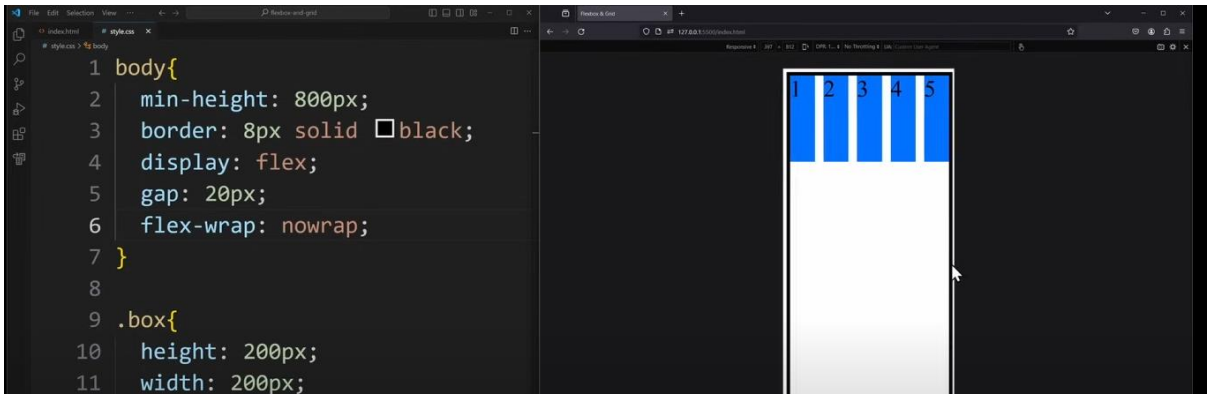
Each flex item can individually override align-items with align-self.



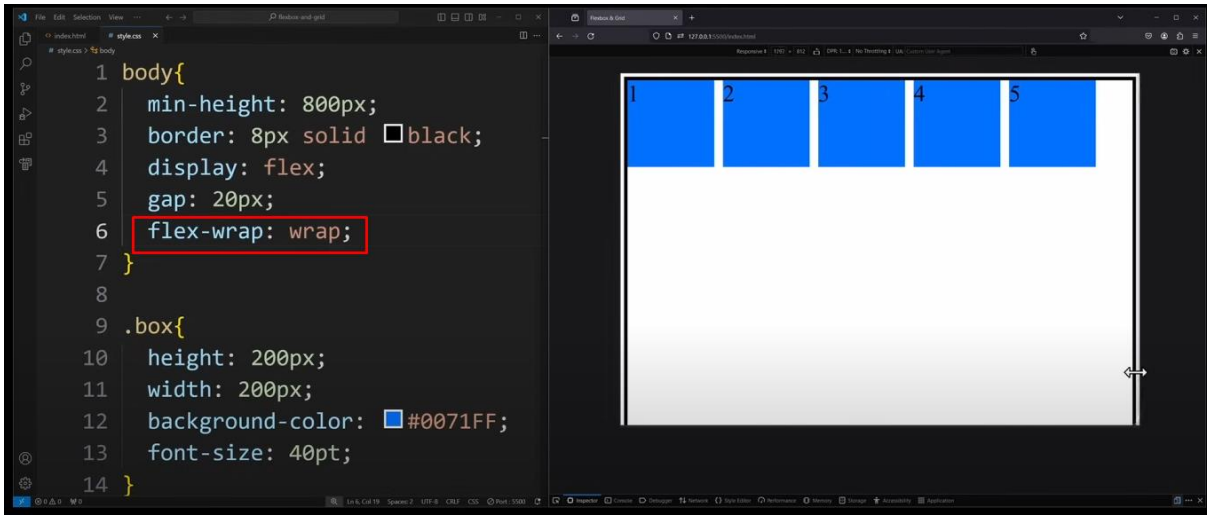
e) flex-wrap (Allows Items to Wrap)

By default, flex items try to fit in one line. flex-wrap controls wrapping behavior.

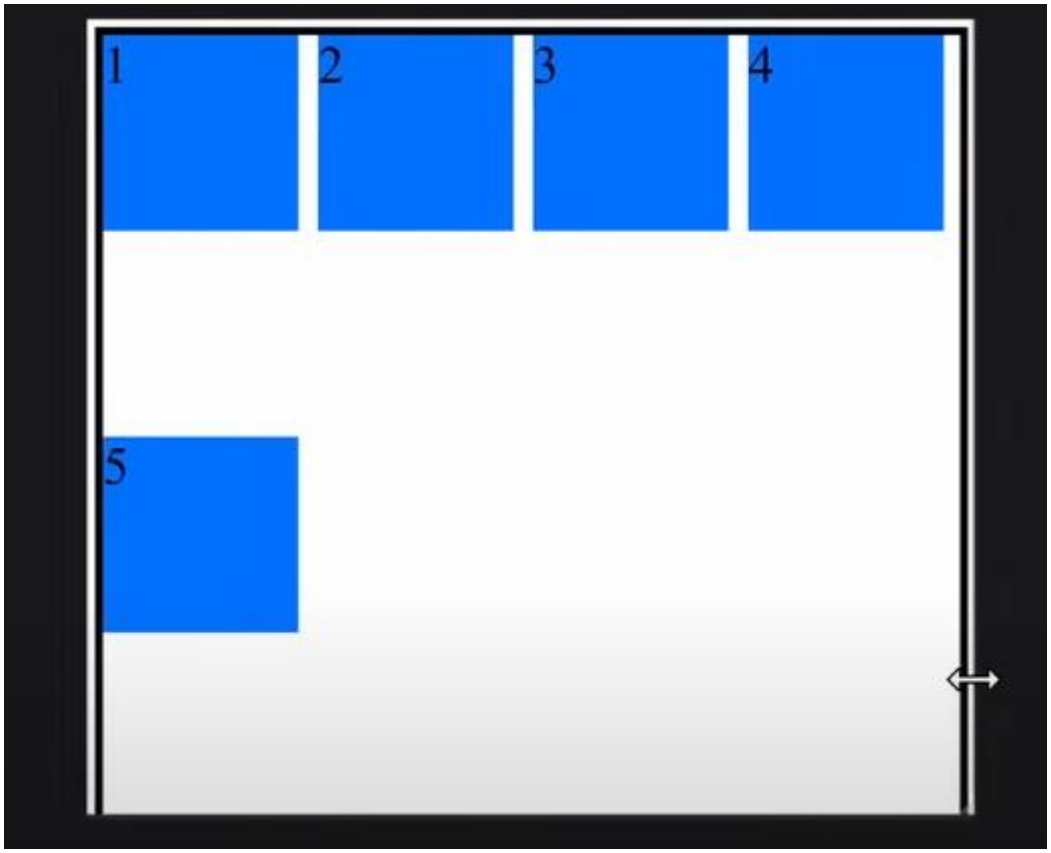
- **nowrap** (default) – Items stay in one line. The boxes may shrink/grow back to original size together as size of window changes



- **wrap** – Items wrap onto multiple lines.



THE 5th element will move to the next row as there is not enough space on the right side of the window



- **wrap-reverse** – Items wrap in reverse order.

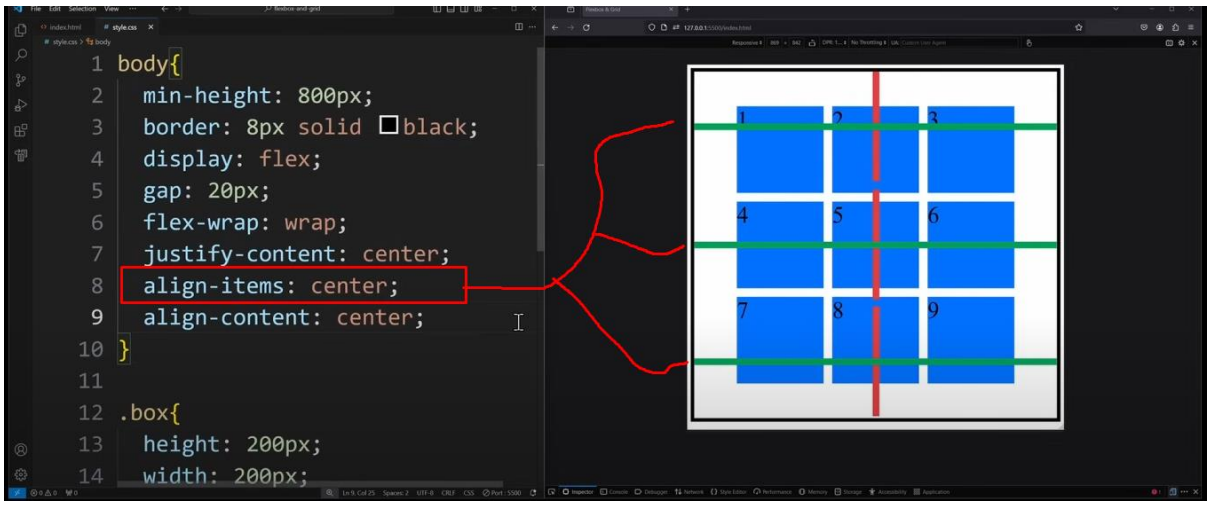
f) align-content (Aligns Wrapped Rows/Columns)

DIFFERENCE BETWEEN ALIGN ITEMS AND ALIGN CONTENT

Both properties control alignment in Flexbox but serve different purposes:

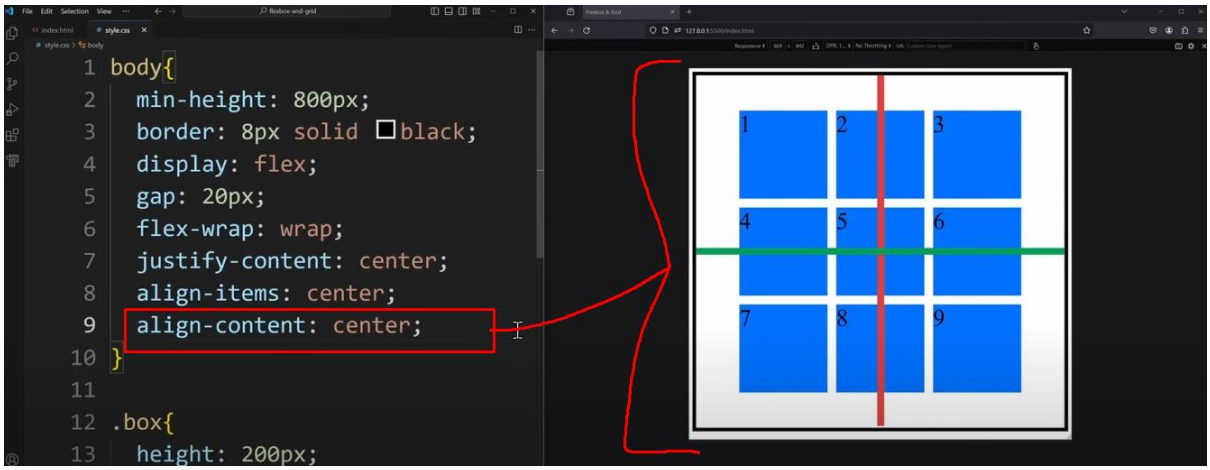
align-items (Aligns Items Within a Single Row/Column)

- Works on the **cross-axis** (perpendicular to flex-direction).
- Controls how **individual flex items** are positioned **within their flex container**.
- Affects items **inside** a single flex line.



align-content (Aligns Multiple Rows/Columns)

- Works when flex-wrap: wrap; is enabled.
- Controls spacing **between multiple flex lines** inside a container.
- Has no effect if items fit in a single row/column.



Affects multiple lines when flex-wrap is applied. It works like justify-content but for wrapped items along the cross axis.

g) flex-grow (Defines How Items Expand)

Determines how much an item grows relative to others.

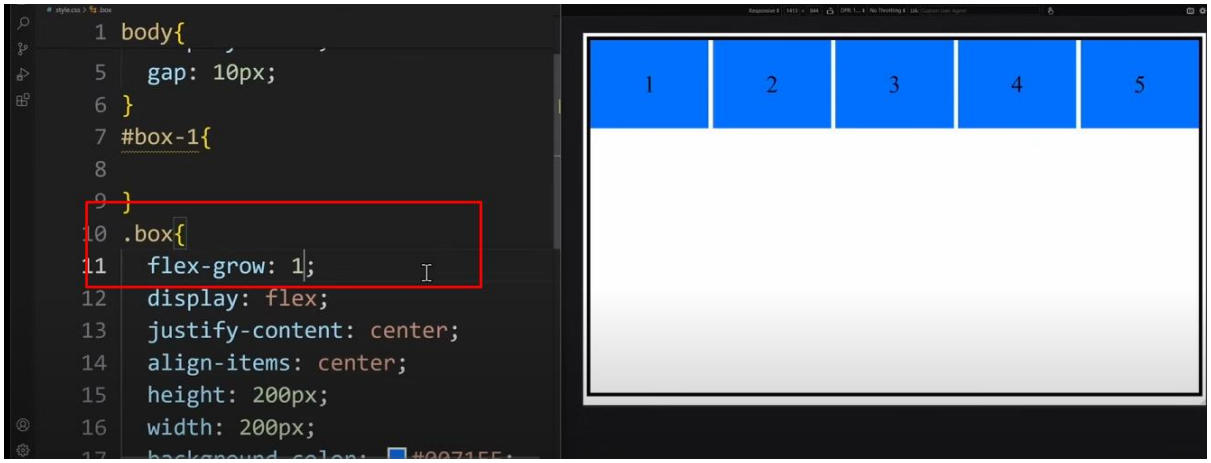
The flex-grow property determines how much a flex item **expands** relative to others when extra space is available inside a flex container.

How It Works

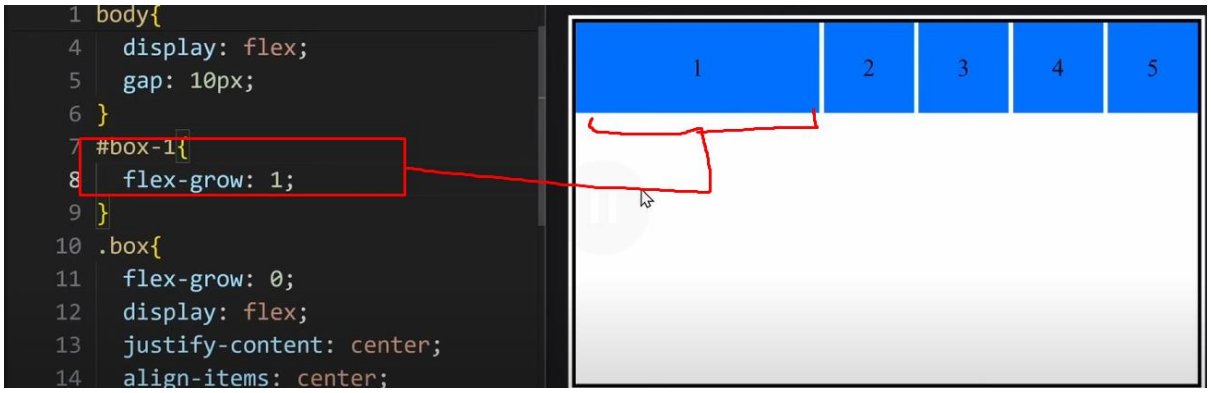
- Each flex item is assigned a **flex-grow value (a number)**.
- The extra space is distributed based on the ratio of flex-grow values among the flex items.
- If all items have flex-grow: 1;, they expand equally.
- If one item has flex-grow: 2; while others have flex-grow: 1;, it takes **twice as much** space as the others.

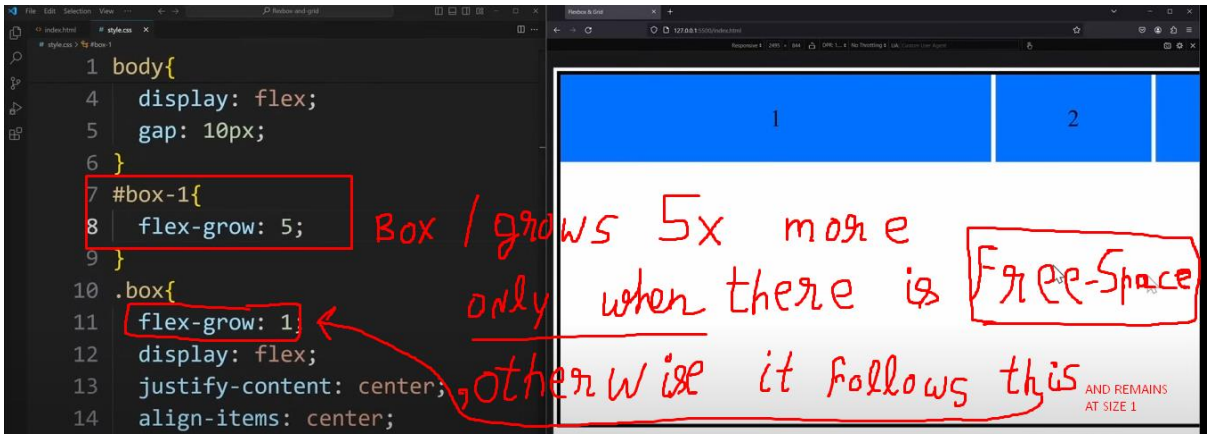
Important Notes:

- If no extra space is available, flex-grow does nothing.
- Default value: flex-grow: 0; (items won't grow unless explicitly set).

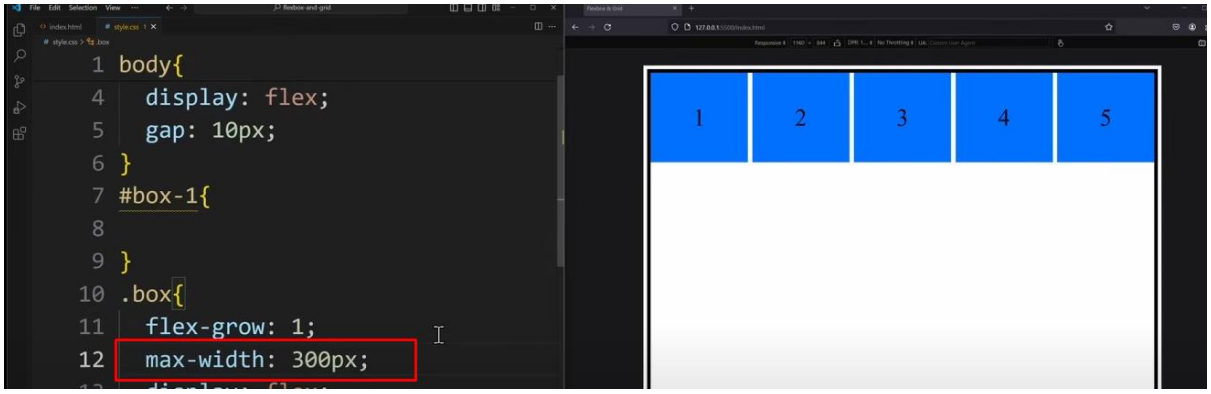


You can also specify which particular element you want to apply the grow property to.





You can also set a max size in units for the element to grow to and then stop at by using max-width

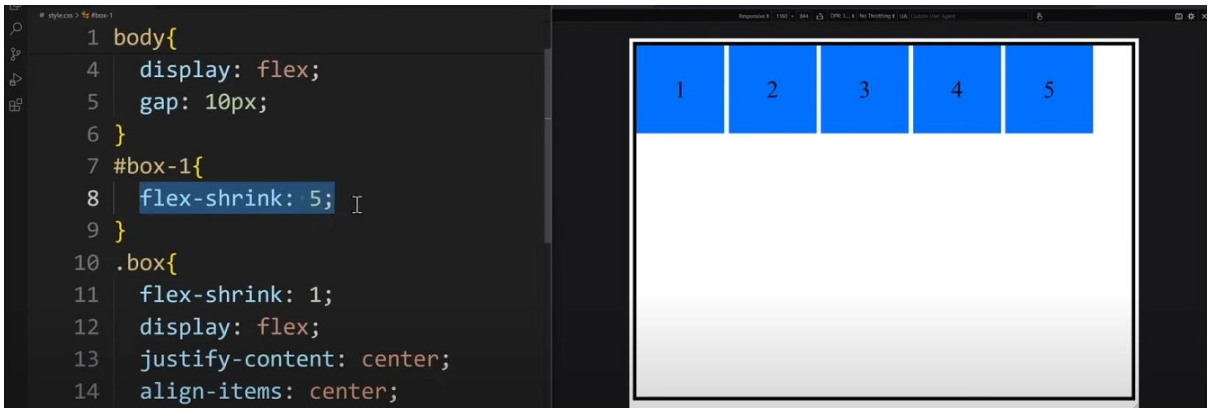


h) flex-shrink (Defines How Items Shrink)

Controls how much an item shrinks if space is limited.

flex-shrink (Reducing Items When Space is Limited)

The flex-shrink property determines how much a flex item **shrinks** when there isn't enough space in the container.



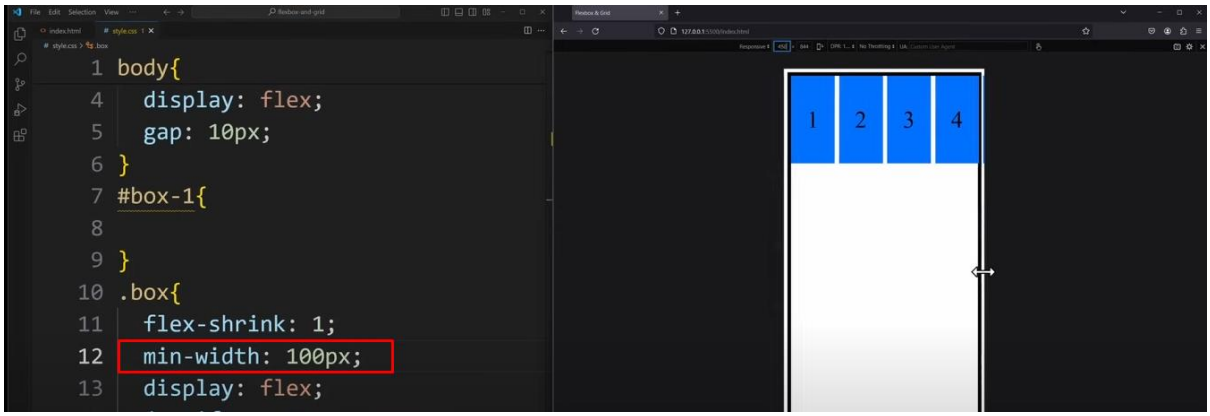
How It Works

- If all items have flex-shrink: 1;, they shrink equally when the container gets smaller.
- If one item has flex-shrink: 2;, it shrinks **twice as fast** as items with flex-shrink: 1;.
- If flex-shrink: 0;, the item **won't shrink** at all, possibly causing overflow.

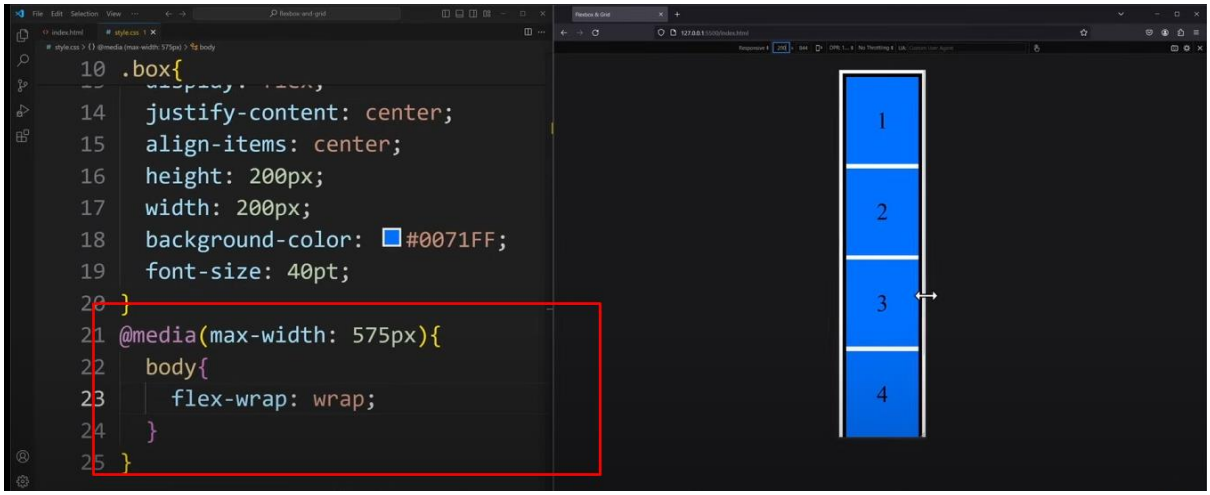
Important Notes:

- The **actual shrinkage** depends on the item's flex-basis (initial size).
- Default value: flex-shrink: 1; (items shrink equally by default).

You can set a minimum value for the elements to shrink to and then stop by using min-width



To solve overflow issues in Flexbox, use flex-wrap: wrap; along with media queries. This ensures items move to the next line instead of shrinking too much or overflowing. On larger screens, flex-wrap: nowrap; keeps items in a single row. For smaller screens, switch to flex-wrap: wrap; in a media query to allow items to stack. Combining flex-wrap with min-width and flex-basis helps maintain a balanced layout across different screen sizes, preventing elements from getting too small or breaking the design.

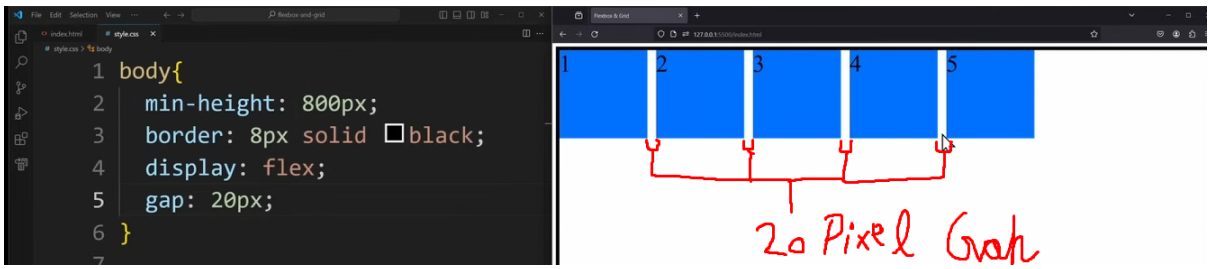


i) flex-basis (Defines Initial Size Before Resizing)

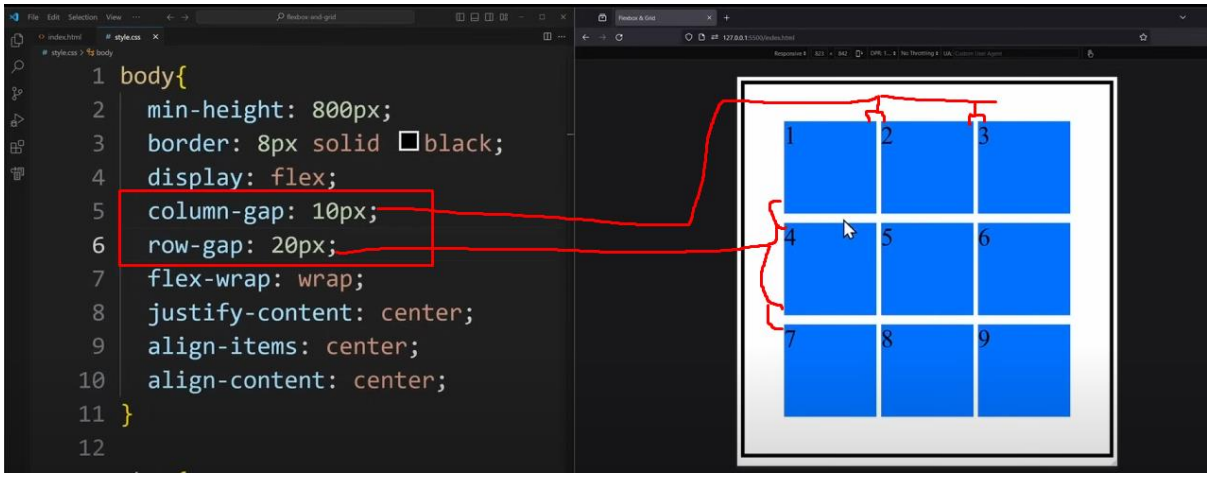
Sets the initial size of an item before flex-grow or flex-shrink takes effect.

j) gap (Spacing Between Items)

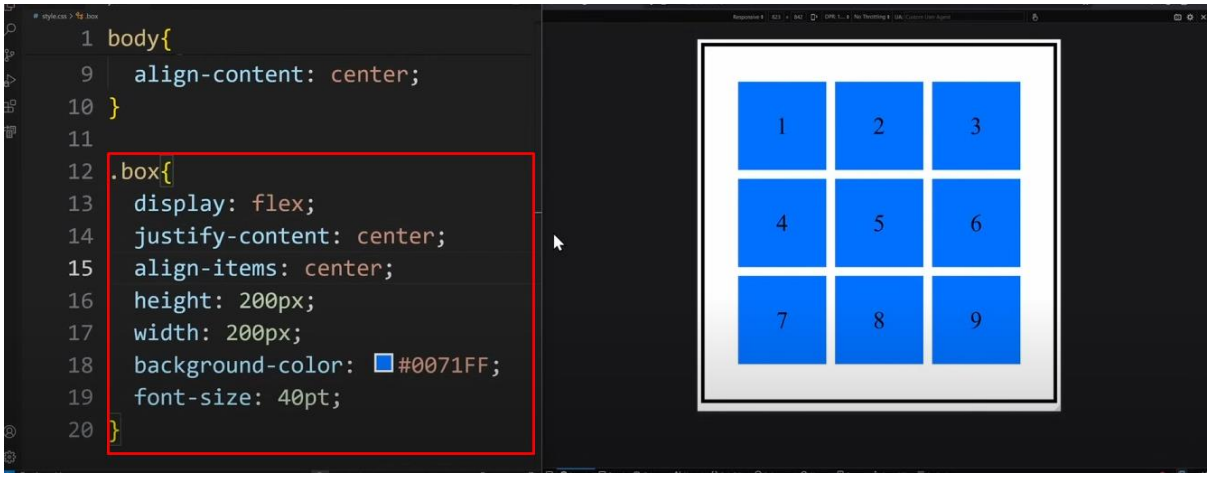
Adds spacing between flex items without using margins.



You can also split the gap into two sub properties : row-gap and column-gap



You can also use flexbox to center elements like the numbers inside the box and not just the body.



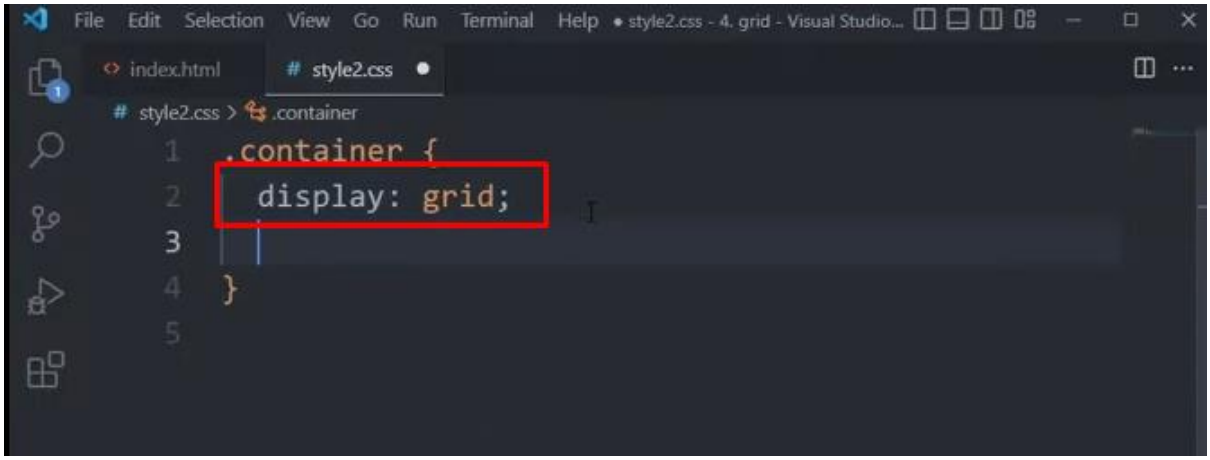
GRID

CSS Grid is a two-dimensional layout system in CSS that provides precise control over **rows and columns**. Unlike Flexbox (which only handles one dimension at a time), Grid enables efficient layout structures for complex web designs.

1. Understanding the Grid System

Grid Container

To create a grid layout, set display: grid; or display: inline-grid; on a container. This element becomes the **grid container**, and all direct children inside it are **grid items**.



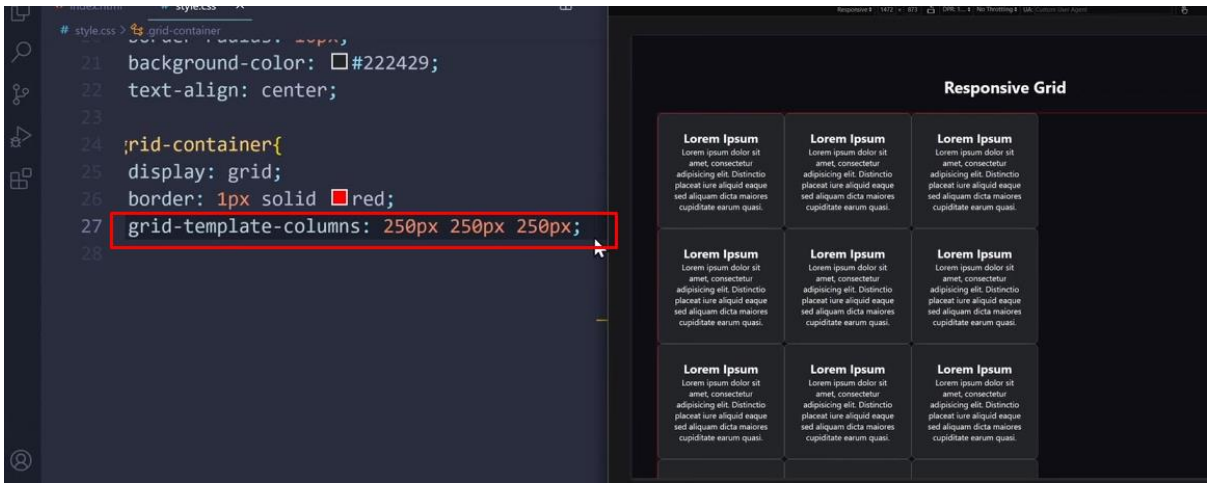
Grid Items

Each element inside a grid container is called a **grid item**. These items can be placed and sized within the grid using various properties.

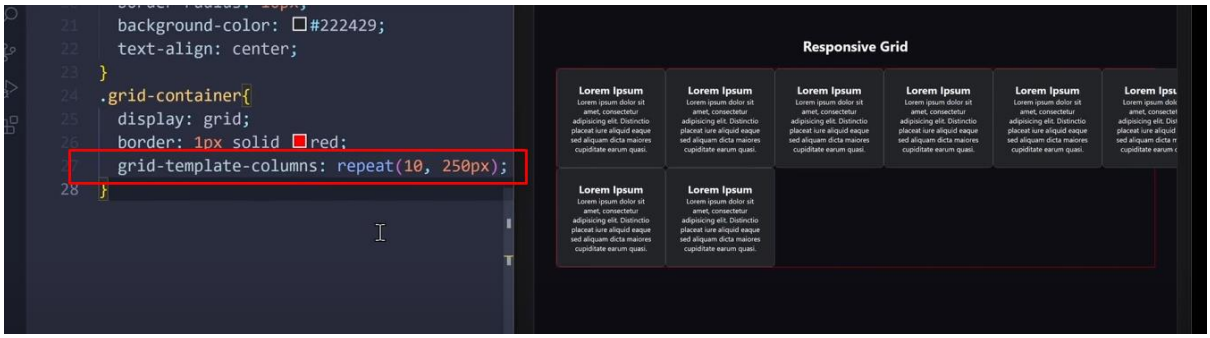
2. Defining Rows & Columns

Grid allows defining **rows and columns** explicitly:

- **grid-template-columns** → Defines the number and size of columns.

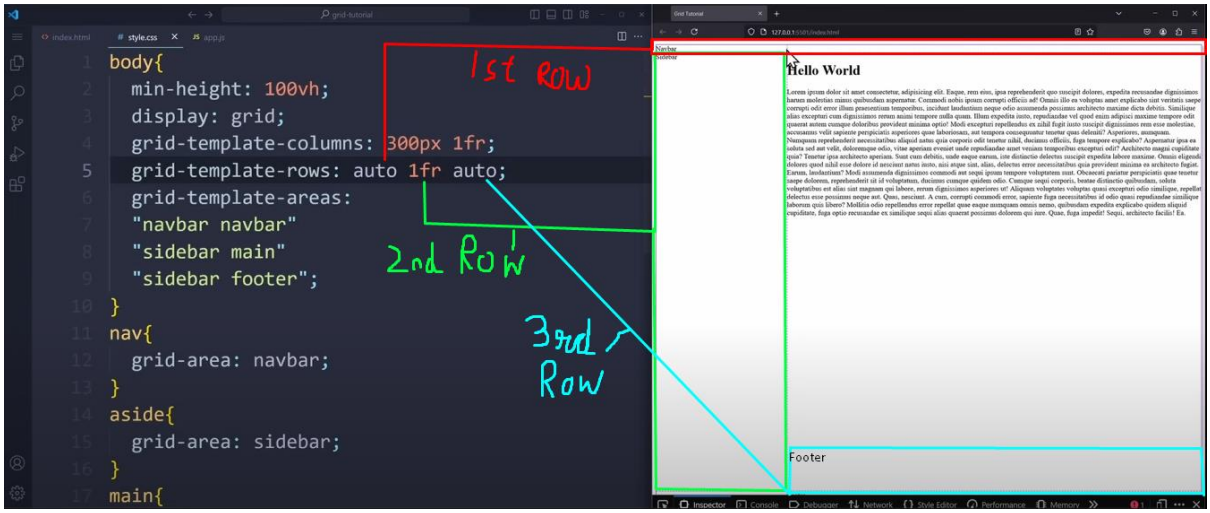


If we want to define the size of a lot of columns it might become tedious so we can use repeat(number of columns , size)



These procedures also apply to rows

- **grid-template-rows** → Defines the number and size of rows.

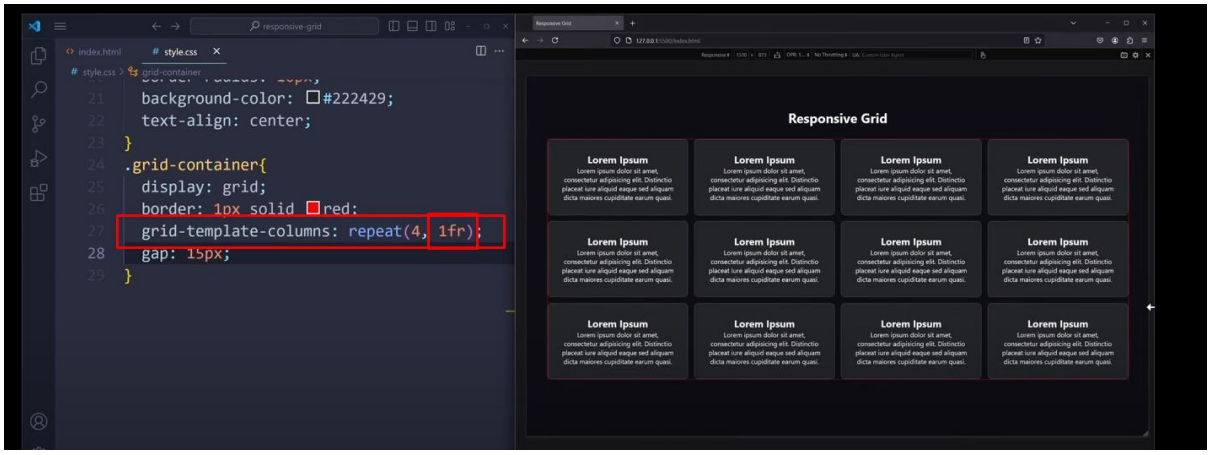


- **grid-template-areas** → Assigns names to sections of the grid, making layouts more readable.

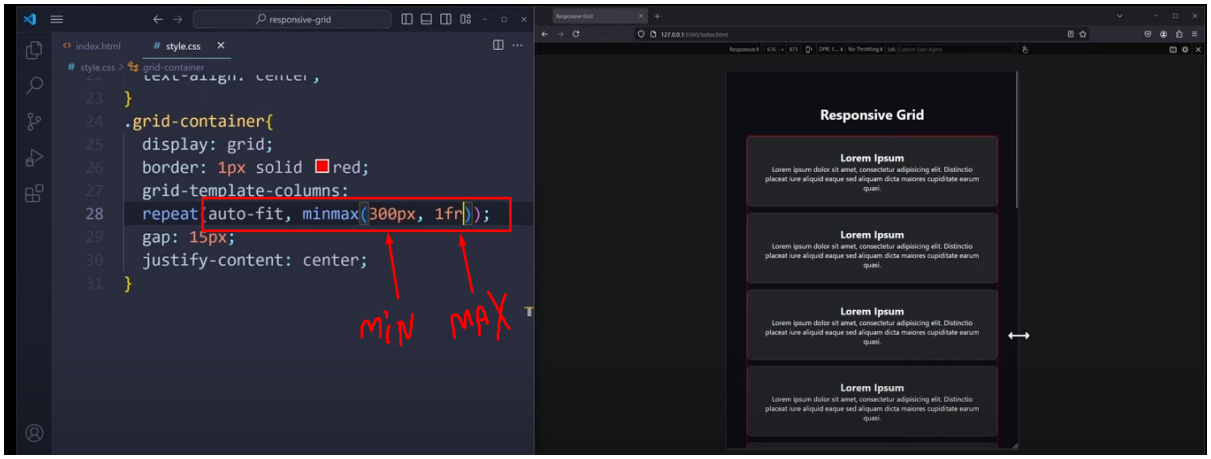
Units Used in Grid

Grid layouts can be defined using:

- **Pixels (px)** → Fixed size (not responsive).
- **Percentages (%)** → Relative to the container.
- **Fractional units (fr)** → Distributes space dynamically.



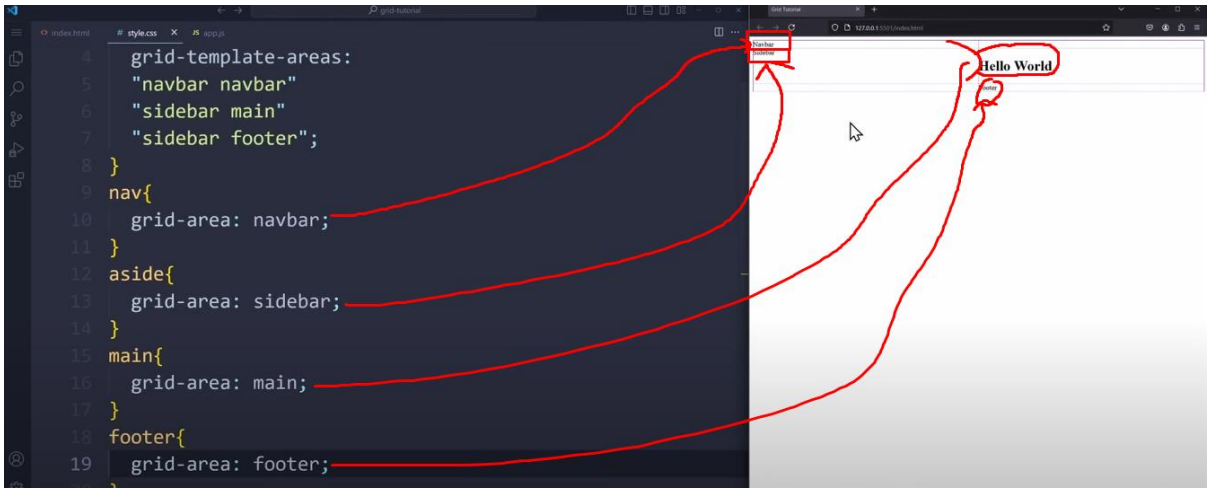
- **auto** → Adjusts to content size.
- **minmax(min, max)** → Defines a flexible range for column or row sizes.



3. Placing Items in the Grid

Grid items can be placed precisely within rows and columns using:

- **grid-column-start / grid-column-end** → Defines how many columns an item should span.
- **grid-row-start / grid-row-end** → Defines how many rows an item should span.
- **grid-area** → Assigns an item to a specific named area.



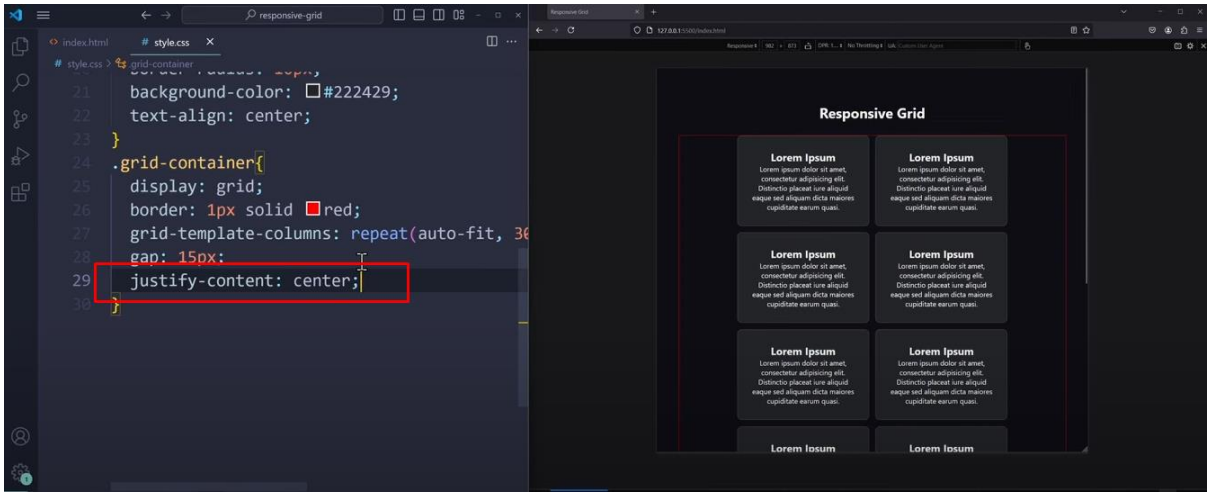
4. Aligning Items in the Grid

Aligning Individual Grid Items

- **justify-self** → Aligns an item horizontally in its grid cell (start, center, end, stretch).
- **align-self** → Aligns an item vertically in its grid cell (start, center, end, stretch).

Aligning the Whole Grid Layout

- **justify-items** → Aligns all items horizontally within their cells.
- **align-items** → Aligns all items vertically within their cells.
- **justify-content** → Aligns the entire grid horizontally within its container.

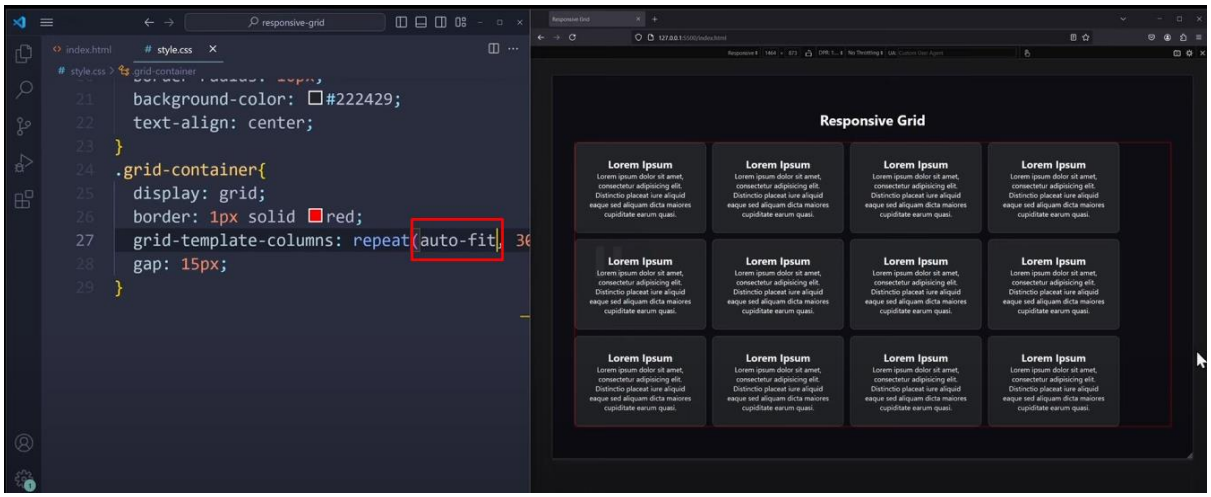


- **align-content** → Aligns the entire grid vertically within its container.

5. Creating Responsive Grids with auto-fit and auto-fill

When creating dynamic layouts, we can use:

- **auto-fit** → Expands items to fill the available space.



- **auto-fill** → Creates as many items as possible without stretching them.

This ensures the grid adapts automatically as the screen size changes.