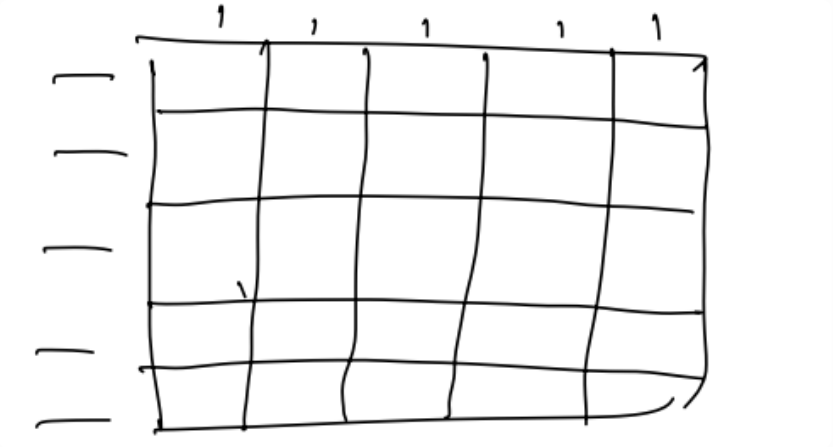
Agenda

Today we will be discussing about Responsive Layout and Grid in CSS.

In previous session, we learned about flexbox and media queries. you can practice more about Flexbox via https://flexboxfroggy.com/.

Sometimes, when you have to create a very complex 2-D layout, you can use a Grid.

Let's take an example to understand this better, when we were kids, there were notebooks and the pages were of below structure.



This is basically a grid structure.

CSS Grid Layout is a layout in CSS that allows you to create two-dimensional grid layouts for arranging and aligning elements on a web page. It provides a powerful way to structure and design both rows and columns of content, allowing for complex and flexible layouts.

In CSS Grid, we can use more properties as compared to Flexbox. So, lets jump on to VS Code to understand CSS Grid better.

Let's create a basic layout of 9 items which we will use to understand Grid.

HTML Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSS Grid</title>
    <style>
```

```
        *{
            padding: 0;
            margin: 0;
        }

        body{
            font-size: 20px;
            line-height: 1.5;
            color: dodgerblue;
            background-color: tomato;
        }

        .container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
        }

        .item{
            background-color: dodgerblue;
            color: white;
            font-size: 20px;
            padding: 20px;
            border: 1px solid lightblue;
        }
    </style>
</head>
<body>
    <div class="container">
        <div class="item">Item 1</div>
        <div class="item">Item 2</div>
        <div class="item">Item 3</div>
        <div class="item">Item 4</div>
        <div class="item">Item 5</div>
        <div class="item">Item 6</div>
        <div class="item">Item 7</div>
        <div class="item">Item 8</div>
        <div class="item">Item 9</div>
    </div>
</body>
</html>
```

Suppose we want to arrange these items in a row order, so what properties can I use?

display: flex;. In last lecture we learn't about flex, so giving display as flex will make the items align in row manner.

Now, what happens if I give display property as Grid?

Nothing will change, they will look the same, but if you go to inspect, you can see a Grid keyword to the container.

```
<div class="container"> grid == $0
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
  <div class="item">Item 6</div>
  <div class="item">Item 7</div>
```

It tells that this container is assigned to a grid value, so we are allowed to use any grid property available with the CSS.

grid-template-columns

So lets move to the first property called **grid-template-columns** and give some value to this as grid-template-columns

```
.container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
            display: grid;
            grid-template-columns: 100px 100px 100px;
        }
```

Here grid will be with three columns, each with a width of 100px. This means that the entire grid will be 300px wide, and each of the nine grid items will occupy a cell that is 100px wide having 3 rows of the grid.

Similarly if we remove 1 column of 100px (grid-template-columns: 100px 100px) from the above property, it will create 2 columns of 100px each.

But, there is a catch, suppose I gave border to the container, Border will be applied to the entire container.

```
.container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
            display: grid;
            grid-template-columns: 100px 100px 100px;
            border: 2px solid black;
        }
```

So here 300px will be taken by the items but the rest 900px will be completely empty. So we are not making any good use of the space. To fix this Grid provides Frames to handle this.

So I give the units like this:

```
.container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
            display: grid;
            grid-template-columns: 1fr 1fr 1fr;
            border: 2px solid black;
        }
```

It will take the entire space and equally divide 3 items in a row. The grid-template-columns: 1fr 1fr 1fr; declaration means that the available space in the container's width will be divided into three equal parts, each represented by 1fr. This is a flexible way to distribute space evenly among the columns.

Let's understand it that way:

- suppose the container is of length 120px.
- grid-template-columns: 1fr 1fr 1fr;
- Total frames will be 1+1+1 = 3.
- size of each frame will be 120/3 = 40px.

Hence in this case, each column of the three columns will be equal to 40px.

Now lets give the units like this:

```
.container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
            display: grid;
            grid-template-columns: 1fr 2fr 1fr;
            border: 2px solid black;
        }
```

Now with grid-template-columns: 1fr 2fr 1fr;, you are distributing the available space into three columns with different proportions. The first column will take up 1fr that is 300px of the available space, the second column will take up 2fr that is 600px, and the third column will take up 1fr or 300px.

column-gap

Now if we want to give some gap between the columns, we can use **column-gap** property

```
.container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
            display: grid;
```

```
            grid-template-columns: 1fr 2fr 1fr;
            column-gap: 20px;
        }
```

The space between the columns will now be 20px, creating a gap between each column. This space will be added to the defined widths of the columns.

row-gap

Similarly if we want to give some gap between the rows, we can use **row-gap** property

```
.container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
            display: grid;
            grid-template-columns: 1fr 2fr 1fr;
            column-gap: 20px;
            row-gap: 20px;
        }
```

The space between the rows and columns will now be 20px. This space will be added to the defined widths of the rows.

gap

If you want to give gap to both rows and columns at once. you can give gap property only.

```
.container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
            display: grid;
            grid-template-columns: 1fr 2fr 1fr;
            gap: 20px;
        }
```

The result will be the same when we used both row-gap and column-gap property.

Let's now see row-level property of grid.

grid-template-rows

Lets give some value to this as grid-template-rows.

```
.container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
            display: grid;
```

```
            grid-template-columns: 1fr 2fr 1fr;
            grid-template-rows: 1fr 2fr 1fr;
        }
```

The **grid-template-rows: 1fr 2fr 1fr;** declaration defines the height proportions of the grid's rows. The first row will take up 1fr, the second row will take up 2fr, and the third row will take up 1fr.

Similar to how fr units distribute space among columns, here the vertical space is distributed among rows. The second row will be twice as tall as the first and third rows.

grid-template-columns with repeat()

Suppose you want to divide each column in equal frames then we can use **repeat()** method.

Suppose you want to create a grid with 4 columns, each of 100px width. Instead of writing out 100px 100px 100px 100px, you can use the repeat() function:

```
.container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
            display: grid;
            grid-template-columns: repeat(3, 1fr);
            grid-template-rows: 1fr 2fr 1fr;
        }
```

This method can also be applied to row level grid.

grid-auto-rows

grid-auto-rows is a CSS property used in CSS Grid layouts to define the size of rows that are created implicitly (automatically) when the content doesn't fit into the explicitly defined rows. This property allows you to control the height of these automatically generated rows.

Suppose we give its value:

```
.container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
            display: grid;
            grid-auto-rows: 200px
        }
```

In this example, all the rows will have a fixed height of 200px. This is helpful while we want out grid to not dynamically grow according to the content.

We can also use all the flex properties with Grid like align-items, justify-content etc.

If I use align-items: end;

```
.container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
            display: grid;
            align-items: end;
        }
```

All the columns will be pushed to the end of the container.

Now we will be working on item level.

Suppose we want to get a particular item without making its another class or id. then we can use **.item:nth-of-type()** pseudo-selector.

Here suppose we want to change the styles particularly for 2nd items from out list, we can use it like:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSS Grid</title>
    <style>
        *{
            padding: 0;
            margin: 0;
        }

        body{
            font-size: 20px;
            line-height: 1.5;
            color: dodgerblue;
            background-color: tomato;
        }

        .container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
            display: grid;
            grid-template-columns: 1fr 1fr 1fr;
            grid-template-rows: 200px 200px 200px;

        }

        .item{
            background-color: dodgerblue;
            color: white;
            font-size: 20px;
```

```
            padding: 20px;
            border: 1px solid rgb(114, 118, 119);
        }

        .item:nth-of-type(2){
            width: 100px;
            height: 100px;
        }
    </style>
</head>
<body>
    <div class="container">
        <div class="item">Item 1</div>
        <div class="item">Item 2</div>
        <div class="item">Item 3</div>
        <div class="item">Item 4</div>
        <div class="item">Item 5</div>
        <div class="item">Item 6</div>
        <div class="item">Item 7</div>
        <div class="item">Item 8</div>
        <div class="item">Item 9</div>
    </div>
</body>
</html>
```

We have selected item 2 and gave its height and width as 100px.

Now, we can give some properties to it:

```
.item:nth-of-type(2){
            width: 100px;
            height: 100px;
            align-self: center;
            justify-self: center;
        }
```

These properties are used to control how the element aligns within its grid cell both vertically and horizontally. align-self: center; centers the element vertically, and justify-self: center; centers it horizontally within the grid cell.

These were the flex properties, all the flex properties are available for CSS grids also.

Keep in mind that these styles will only be applied to the second .item element due to the :nth-of-type(2) selector. The other .item elements will not be affected by these specific styles unless they are targeted by other rules.

Now, let's take 1 more example, where you want item 1 to take space of item 4 vertically. So how we can do this?

We can give properties like:

```
.item:nth-of-type(1){
            background: black;
            grid-row: 1/3;
        }
```

This will create a lot of confusion that 1/3 row is how that much. So for that we will open inspect and see breakpoints.

You can see every element has breakpoint, so we are saying that 1st element should row-wise take the 1st to 3rd breakpoint.

grid-row: 1/3; sets the vertical span (row range) that the targeted .item element should occupy. In this case, the .item will span from row line 1 to row line 3.

If you make it grid-column: 1/3; it will horizontally take the space from 1st breakpoint to starting of 3rd breakpoint.

There is 1 easier way to do this, you can give grid-column: span 2; and it will do the same thing.

```
.item:nth-of-type(1){
            background: black;
            grid-column: span 2;
        }
```

The grid-column: span 2; property on the .item class indicates that each .item element will span two columns horizontally.

Okay, so now we will move ahead with media queries and us it with Grids in CSS.

Media queries is used to make your webpage responsive. Media queries allows you to apply different styles based on the characteristics of the or screen, such as its width, height, orientation, resolution, and more. Media queries enable you to create responsive designs that adapt to various screen sizes and devices.

Lets just check the responsiveness of our basic code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSS Grid</title>
    <style>
        *{
            padding: 0;
            margin: 0;
        }
```

```
        body{
            font-size: 20px;
            line-height: 1.5;
            color: dodgerblue;
            background-color: tomato;
        }

        .container{
            width: 1200px;
            margin: 100px auto;
            padding: 10px;
            display: grid;
            grid-template-columns: 1fr 1fr 1fr;
            border: 2px solid black;
        }

        .item{
            background-color: dodgerblue;
            color: white;
            font-size: 20px;
            padding: 20px;
            border: 1px solid lightblue;
        }
    </style>
</head>
<body>
    <div class="container">
        <div class="item">Item 1</div>
        <div class="item">Item 2</div>
        <div class="item">Item 3</div>
        <div class="item">Item 4</div>
        <div class="item">Item 5</div>
        <div class="item">Item 6</div>
        <div class="item">Item 7</div>
        <div class="item">Item 8</div>
        <div class="item">Item 9</div>
    </div>
</body>
</html>
```

Lets try to downsize the screen and move the screen. You can see that this webpage is not responsive and when we downsize it part of grid items is vanishing out of the window.

So, to make it responsive we will be using media queries.

Similarly, we can check this window using inspect responsive feature. Its not responsive there also, so we have to make it responsive.4

Suppose, we are downsizing the screen and want to it make responsive as soon as screen width reaches 500px. We can use the syntax as:

```
@media (max-width: 500px){
    .container{
        grid-template-columns: 1fr;
```

```
    }
}
```

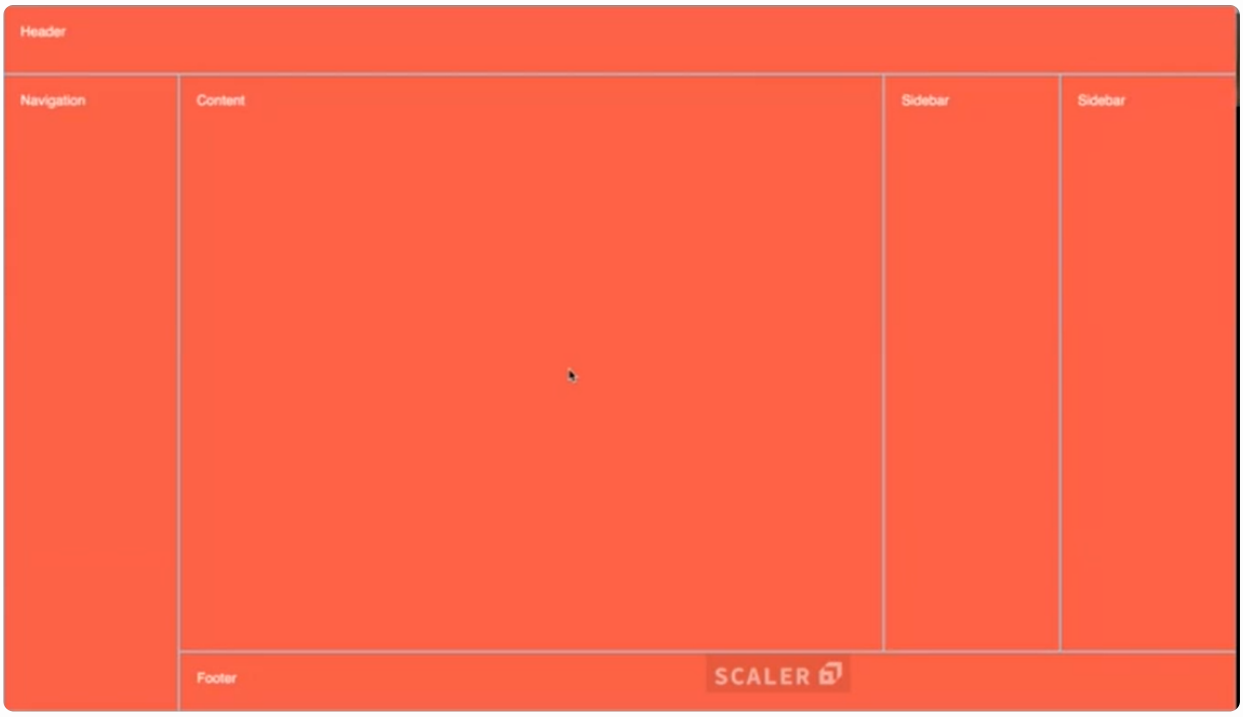
Now lets see what happens as we downsize the window.

- When the screen width is 500px or narrower, the styles inside the media query will be applied.
- The .container element's grid layout will be set to a single column using grid-template-columns: 1fr;.
- This means that on screens narrower than 500px, the grid will collapse into a single column layout. Each grid item will take up the full width of the container, allowing for easy vertical scrolling on smaller screens.

When screen size is reaching 500px, we are telling the grid to make only 1 column and take the whole frame size for each item.

So media queries is applied this this, now we will cover 1 more topic from Grid that is grid areas.

For understanding the Grid Areas, we have to make the layout of the screen in this exercise:



Lets first define all the components in our HTML page, header, navigation, sidebar, footer.

For all these components we can create semantic tags for all.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Grid Layout</title>
</head>
<style>
    *{
        margin: 0;
        padding: 0;
    }

    body{
        font-size: 20px;
        height: 100vh;
    }
</style>
<body>
    <header>Header</header>
    <main>Content</main>
    <nav>Navigation Bar</nav>
    <aside>SideBar</aside>
    <footer>footer</footer>
</body>
</html>
```

So, now we have created a basic layout of the application.

for initializing grid in our application, we can use display: grid; property.

```
body{
        font-size: 20px;
        height: 100vh;
        display: grid;
    }
```

now lets give some borders and colors to all the elements.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Grid Layout</title>
</head>
<style>
    *{
        margin: 0;
        padding: 0;
    }

    body{
        font-size: 20px;
        height: 100vh;
        display: grid;
    }

    header, main, nav, aside, footer{
```

```
        background-color: tomato;
        color: white;
        padding: 20px;
        border: skyblue 1px solid;
    }
</style>
<body>
    <header>Header</header>
    <main>Content</main>
    <nav>Navigation Bar</nav>
    <aside>SideBar</aside>
    <footer>footer</footer>
</body>
</html>
```

To convert this layout to our desired layout, we can use 1 css property called grid-template-areas, where we can tell CSS what area to give to a specific section.

we can use grid-area to call that specific area and provide that area to our elements

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Grid Layout</title>
</head>
<style>
    *{
        margin: 0;
        padding: 0;
    }

    body{
        font-size: 20px;
        height: 100vh;
        display: grid;
        grid-template-areas:
        'top top top'
        'nav content sidebar'
        'nav footer footer';
    }

    header, main, nav, aside, footer{
        background-color: tomato;
        color: white;
        padding: 20px;
        border: skyblue 1px solid;
    }

    header{
        grid-area: top;
    }

    nav{
        grid-area: nav;
```

```
    }

    main{
        grid-area: content;
    }

    aside{
        grid-area: sidebar;
    }
    footer{
        grid-area: footer;
    }
</style>
<body>
    <header>Header</header>
    <main>Content</main>
    <nav>Navigation Bar</nav>
    <aside>SideBar</aside>
    <footer>footer</footer>
</body>
</html>
```

The CSS styles define the grid layout using grid-template-areas, specifying where each section should appear in the grid. The classes header, nav, main, aside, and footer are assigned to their respective grid areas using the grid-area property.

Now using this code we have reached the basic layout requirements for us. But we have to play with heights and widths of the elements to make it exactly like the desired one.

We first need to add 1 more sidebar for our requirements.

Ans,

- we need to add 4 templates in grid-template-areas property,
- we have to make 1 more element for sidebar and give it the same styling
- then we can add new template name to grid-template-areas and create new element style with respect to our template.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Grid Layout</title>
</head>
<style>
    *{
        margin: 0;
        padding: 0;
    }

    body{
        font-size: 20px;
```

```
        height: 100vh;
        display: grid;
        grid-template-areas:
        'top top top top'
        'nav content sidebar sidebar2'
        'nav footer footer footer';
    }

    header, main, nav, aside, footer, div{
        background-color: tomato;
        color: white;
        padding: 20px;
        border: skyblue 1px solid;
    }

    header{
        grid-area: top;
    }

    nav{
        grid-area: nav;
    }

    main{
        grid-area: content;
    }

    aside{
        grid-area: sidebar;
    }
    footer{
        grid-area: footer;
    }
    div{
        grid-area: sidebar2;
    }
</style>
<body>
    <header>Header</header>
    <main>Content</main>
    <nav>Navigation Bar</nav>
    <aside>SideBar</aside>
    <div>Sidebar 2</div>
    <footer>footer</footer>
</body>
</html>
```

Now only height and width of the templates is left, so we can use grid-template-columns and grid-template-rows properties:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Grid Layout</title>
```

```
</head>
<style>
    *{
        margin: 0;
        padding: 0;
    }

    body{
        font-size: 20px;
        height: 100vh;
        display: grid;
        grid-template-areas:
        'top top top top'
        'nav content sidebar sidebar2'
        'nav footer footer footer';
        grid-template-columns: 1fr 4fr 1fr 1fr;
        grid-template-rows: 80px 1fr 70px;
    }

    header, main, nav, aside, footer, div{
        background-color: tomato;
        color: white;
        padding: 20px;
        border: skyblue 1px solid;
    }

    header{
        grid-area: top;
    }

    nav{
        grid-area: nav;
    }

    main{
        grid-area: content;
    }

    aside{
        grid-area: sidebar;
    }
    footer{
        grid-area: footer;
    }
    div{
        grid-area: sidebar2;
    }
</style>
<body>
    <header>Header</header>
    <main>Content</main>
    <nav>Navigation Bar</nav>
    <aside>SideBar</aside>
    <div>Sidebar 2</div>
    <footer>footer</footer>
</body>
</html>
```

- **grid-template-columns:** 1fr 4fr 1fr 1fr; defines the widths of the columns in the grid. The first column takes up 1 fraction of the available space, the second column takes up 4 fractions, and the last two columns take up 1 fraction each.
- **grid-template-rows:** 80px 1fr 70px; defines the heights of the rows in the grid. The first row has a fixed height of 80px, the second row takes up 1 fraction of the available space, and the third row has a fixed height of 70px.

In this, we have achieved the layout that we wanted to achieve.

Combining Fixed and Flexible Sizes

1. Mixing Fixed and Auto: You can mix fixed widths with auto to create grids where some columns have a specific size, and others adjust based on content size. For example

```
grid-template-columns: 100px auto 200px;
```

This sets up a grid with three columns where the first is 100px wide, the second adjusts automatically, and the third is 200px wide.

4. Using fr Unit for Flexible Columns: The fr unit represents a fraction of the available space in the grid container. It's great for creating flexible grids where columns take up a portion of the space.

For example:

```
grid-template-columns: 1fr 2fr 1fr;
```

This creates a three-column grid where the middle column is twice as wide as the side columns.

Advanced Patterns with repeat()

Combining repeat() with Fixed and Flexible Sizes: You can use repeat() alongside fixed sizes and the fr unit. For instance:

```

```

This creates a grid with three columns: two flexible columns (each taking up an equal fraction of the available space) and a third 100px wide column.

Creating Complex Grids: You can create more complex patterns by repeating a set of column sizes. For example:

```

```

This repeats the pattern 1fr 2fr three times, resulting in a six-column grid with alternating column sizes.

Flexbox vs Grid

Flexbox One-Dimensional Layout: Flexbox is primarily used for laying out items in a single dimension, either a row or a column. Content-First Approach: It's ideal when the size of the flex container's content or the number of items is dynamic or unknown. Alignment and Distribution: Easily align items vertically or horizontally and distribute space between items.

Key Properties:

1. `display: flex` **or** `display: inline-flex` **:** Defines a flex container.
2. `flex-direction` **:** Specifies the direction of the main axis (row, row-reverse, column, column-reverse).
3. `justify-content` **:** Aligns items along the main axis.
4. `align-items` **:** Aligns items along the cross axis.
5. `flex-wrap` **:** Defines whether items should wrap into multiple lines or not.

- Use Flexbox when:

- You're aligning a navigation bar, centering an item on the screen, or building a small-scale layout with a linear flow.

- Ideal for one-dimensional layouts, such as navigation bars, headers, footers, and simple component arrangements.

- Suited for aligning items within a container along a single axis.

- Great for creating responsive designs when dealing with dynamic content.

Grid Two-Dimensional Layout: Grid allows for laying out items in both rows and columns simultaneously. Layout-First Approach: Best for when you have a clear idea of the layout, regardless of the content. Grid Areas: Enables creating complex layouts by defining areas in a grid. Consistent Sizing: Offers control over row and column sizing, and the ability to align the entire grid. Complex Layouts: More suitable for larger, more complex layouts.

Key Properties:

1. `display: grid` **:** Defines a grid container.

2. `grid-template-rows` **and** `grid-template-columns` **:** Specifies the size and structure of the grid.
3. `grid-gap` **or** `grid-row-gap` **and** `grid-column-gap` **:** Defines the space between grid items.
4. `grid-template-areas` **:** Assigns names to areas of the grid.
5. `justify-items` **and** `align-items` **:** Aligns grid items within the grid cells.

- Use Grid when:

- Creating a complex web page layout with multiple rows and columns, like a photo gallery or a magazine-style layout,image galleries, and responsive dashboards.

- Well-suited for complex layouts with both rows and columns.

- Provides a powerful solution for defining the placement and alignment of items in a grid.

Resources to Follow -

A. Understanding the Basics

1. **CSS Grid Layout**:
    - Learn the fundamentals of CSS Grid and how it differs from other layout methods.
    - MDN Web Docs: CSS Grid Layout

2. **Grid Container and Items**:
    - Explore the concepts of grid containers and items, understanding their roles in layout creation.
    - CSS-Tricks: A Complete Guide to Grid

B. Creating Grids

1. **Grid Lines and Units**:
    - Understand grid lines and units for defining column and row sizes.
    - CSS Grid: Unit types

2. **Grid Template Areas**:
    - Learn how to use named areas for a more visual and structured grid layout.
    - CSS-Tricks: A Complete Guide to Grid - Template Areas

A. Responsive Design

1. **Media Queries with Grid**:
    - Explore techniques to make your grids responsive using media queries.
    - CSS-Tricks: A Complete Guide to Grid - Media Queries

2. **Auto-Fit and Auto-Fill**:
    - Understand the difference between auto-fit and auto-fill for dynamic grid sizing.
    - CSS-Tricks: Auto-Sizing Columns in CSS Grid: `auto-fill` vs `auto-fit`

B. Grid Layout Techniques

1. **Nested Grids**:
    - Learn how to create complex layouts by nesting grids within grids.
    - CSS-Tricks: Nested Grids

2. **Grid and Flexbox Integration**:
    - Understand how to combine CSS Grid and Flexbox for powerful layout options.
    - CSS-Tricks: Using CSS Grid the Right Way

III. Resources for Practice

A. Interactive Learning Platforms

1. CodePen: Practice CSS Grid by creating and sharing projects with the community.
2. Grid Garden: An interactive game to reinforce grid layout concepts.
3. CSS Grid Layout by Example: Rachel Andrew's comprehensive resource for learning CSS Grid through practical examples.

B. Coding Challenges

1. CSS Grid Challenges on Frontend Mentor: Real-world projects to apply your CSS Grid skills.
2. Flexbox Froggy: While focused on Flexbox, it's a great resource to reinforce general layout concepts.

C. Additional Reading

1. CSS Grid Layout Module Level 1 - W3C Specification: The official W3C specification for CSS Grid.
2. CSS Grid Layout - Jen Simmons: Jen Simmons' blog and video series provide deep insights into CSS Grid.

IV. Recommended Books

- **"CSS Grid Layout: Learn by Example"** by Michael Bright: A hands-on guide with practical examples to master CSS Grid.

Conclusion

By exploring these resources, you'll gain a solid foundation in CSS Grid and be well-prepared for the class. Remember to practice regularly, as hands-on experience is crucial for mastering any web development skill.