

Esse **Projeto** é para você que deseja adquirir conhecimento nas duas principais maneiras de dispor elementos em uma página HTML, ou seja, utilizando o **Flexbox** e o **Grid**.

Você aprenderá esses conceitos praticando em um projeto chamado **Culturama**, uma página para divulgação de eventos culturais. Vamos desenvolvê-lo para três telas principais: mobile, com largura de 360 pixels; tablets, com largura de 720 pixels; e desktops, com largura de 1440 pixels.

Vamos organizar todos os elementos desse projeto, como o cabeçalho, a seção de banner, categorias, destaques, "próximos eventos", "coloque na sua agenda" e o rodapé.

Como nosso foco é praticar Flexbox e Grid, é importante que você já tenha realizado todos os Projetos que estão indicados como pré-requisitos desse treinamento. Dessa forma, você já terá construído uma base sólida e poderá mergulhar nesses estudos.

Após finalizar o Projeto, você conseguirá aplicar os conhecimentos adquiridos de Flexbox e Grid para posicionar os elementos na página, e praticar em diversos outros projetos que vão expandir o seu portfólio.

Vamos lá, que temos muito trabalho pela frente!

## **Olá, dev!**

Desejamos boas vindas ao Projeto Praticando CSS: Grid e Flexbox!

Nesta jornada de aprendizado vamos focar em organizar todos os elementos do projeto **Culturama**, que consiste em uma página de divulgação de eventos culturais, criada especialmente para este Projeto.

Vamos posicionar os elementos unindo as soluções de layout mais recomendadas para obter um resultado responsivo atualmente, que são o Grid e o Flexbox, além disso, vamos desenvolver utilizando da técnica mobile-first.

Você pode [baixar o arquivo zip inicial do projeto](#),

Fique à vontade para acessar o [layout do projeto no Figma](#).

O editor de código utilizado neste Projeto é o [VSCode](#), você também pode baixar a extensão [Live Server](#) utilizada no editor de código.

Que bom que você decidiu aprender Flexbox e Grid conosco! Esse conhecimento certamente será de muita importância na sua carreira como pessoa desenvolvedora em Front-end. Começaremos abrindo o projeto no VS Code, o editor de código que utilizaremos no decorrer desse Projeto.

Vou abrir o projeto inicial no VS Code que será o editor de código utilizado neste Projeto, mas você pode utilizar o editor que preferir. O projeto inicial deste Projeto está disponível na atividade "Preparando o ambiente". Nessa mesma atividade também se encontra o link para acessar o layout do projeto pronto no Figma, caso queira ir conferindo por conta própria.

No editor de código, clicaremos com o botão direito na pasta "style", localizada no menu à esquerda, e então em "New File". Criaremos um arquivo flex.css, dentro do qual vamos inserir todo o código do Flexbox ao decorrer do Projeto.

Na linha 14 do arquivo index.html, usaremos a tag <link> para linkar o arquivo que acabamos de criar, acessado pelo endereço ./assets/style/flex.css.

```
<!DOCTYPE html>
```

```
<html lang="pt-br">
```

```
<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>Culturama</title>
```

```
  <link
```

```
    href="https://fonts.googleapis.com/css2?family=Fjalla+One&family=Work+Sans&display=swap"
    rel="stylesheet">
```

```
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/meyer-
  reset/2.0/reset.min.css"
```

```
    integrity="sha512-
  NmLkDIU1C/C88wi324HBc+S2kLhi08PN5GDeUVVVC/BVt/9Izdsc9SveVfA1UZbY3sHUIDSyRXhCz
  Hfr6hmPPw=="
```

```
    crossorigin="anonymous" referrerpolicy="no-referrer" />
```

```
  <link rel="shortcut icon" href="./assets/img/Favicon2.png" type="image/x-icon">
```

```
  <link rel="stylesheet" href="./assets/style/grid.css">
```

```
  <link rel="stylesheet" href="./assets/style/flex.css">
```

```
  <link rel="stylesheet" href="./assets/style/style.css">
```

```
//código omitido...
```

Note que estamos utilizando a extensão Live Server, que também ficará disponível na atividade "Preparando o Ambiente" caso você queira utilizá-la.

Feito isso, vamos abrir o projeto no navegador. Primeiramente, precisamos organizar o cabeçalho do projeto Culturama. Ele consiste em uma logo e três botões ("Programação", "Categorias" e "Seu local"), além de um campo de pesquisa logo abaixo.



Para organizar esses elementos com o Flexbox, precisamos localizá-los no arquivo HTML. Acessando o index.html, percebemos que nosso <header> (cabeçalho) é composto por uma tag <nav> e uma tag <ul>, dentro da qual temos diversas <li> com os itens em seu interior.

<header>

<nav>

<ul class="menu">

<li class="menu\_\_item">

<h1 class="menu\_\_titulo"></h1>

</li>

<li class="menu\_\_item"><a href="#" class="">Programação</a></li>

<li class="menu\_\_item"> <a href="#">Categorias</a></li>

<li class="menu\_\_item"><a href="#">Seu local</a></li>

<li class="menu\_\_item menu\_\_pesquisar">

<label class="menu\_\_label" for="pesquisar"><input class="menu\_\_input" type="search" name="" placeholder="O que você procura?"

id="pesquisar"></label>

</li>

</ul>

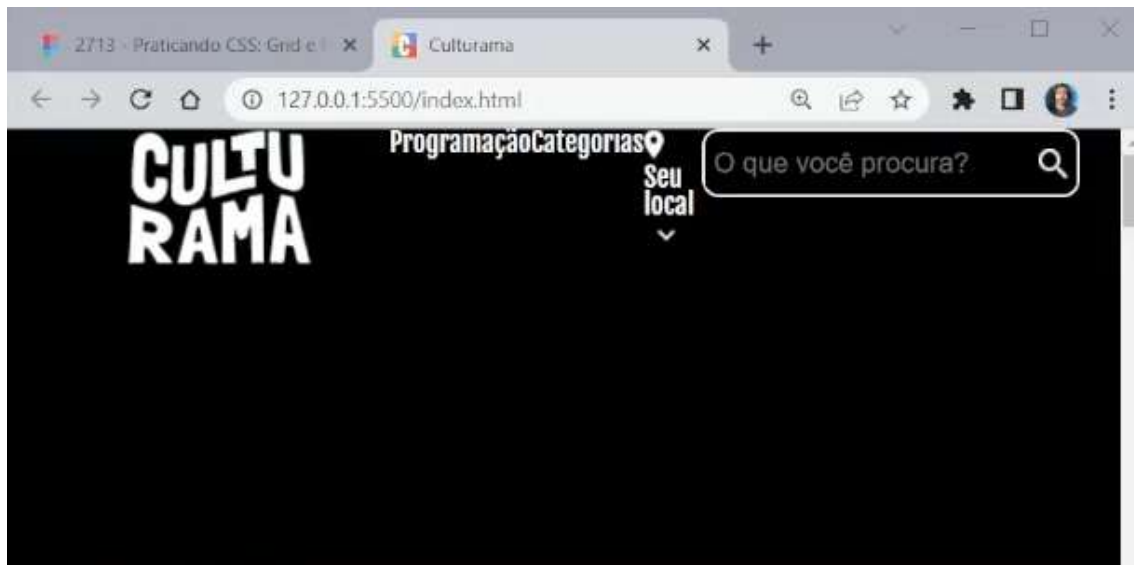
</nav>

</header>

O elemento-pai de todos esses itens é a `<ul>` com a classe `menu`, e é ela que utilizaremos para essa organização. No arquivo `flex.css`, chamaremos a classe `.menu` e aplicaremos a ela, dentro de chaves, a propriedade `display: flex`.

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

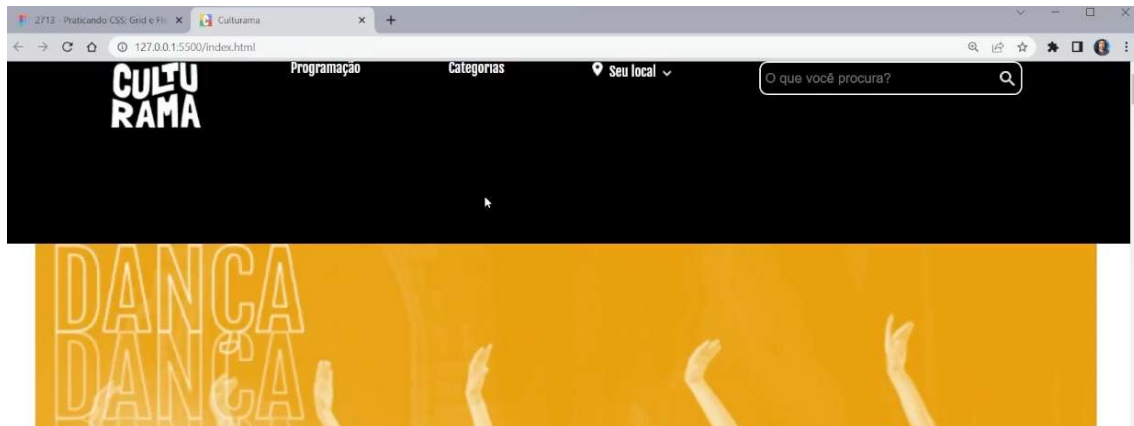
Agora todos os elementos dentro da lista ficaram um ao lado do outro, o comportamento padrão desse `display`. Quando aplicamos o `display: flex` a um elemento, nós o transformamos em um *flex container*, e os elementos em seu interior se tornam *flex items*. A partir daí, podemos manipular esses elementos com diversas propriedades do Flexbox.



Uma delas é a `justify-content`, por exemplo com o valor `space-evenly`. Após atualizar, vemos que foi aplicado um espaçamento horizontal entre todos os elementos, que é aumentado conforme o tamanho da tela também aumenta.

```
.menu {
  display: flex;
  justify-content: space-evenly;
}
```

Essa é uma das características do Flexbox: ele ajuda a organizar os elementos de forma automática e responsiva, se ajustando à largura do layout.



Percebemos que os elementos do cabeçalho estão muito próximos do topo da tela. Para centralizá-los na vertical - ou seja, fora do eixo principal -, utilizaremos a propriedade `align-items` com o valor `center`. Dessa forma, teremos os elementos centralizados na vertical.

```
.menu {
  display: flex;
  justify-content: space-evenly;
  align-items: center;
}
```

Inspecionando a página (algo que fazemos usando o botão F12), vemos que na largura do layout para mobile, 360 pixels, os itens do cabeçalho ficaram muito muito próximos e na mesma linha, dificultando a visualização.

Queremos ter uma quebra de linha, o que conseguiremos aplicando a propriedade `flex-wrap` com valor `wrap`. Agora, quando um elemento não tiver mais espaço pra ficar na mesma linha que os outros, ele será movido ("quebrado") para a linha de baixo.

```
.menu {
  display: flex;
  justify-content: space-evenly;
  align-items: center;
  flex-wrap: wrap;
}
```



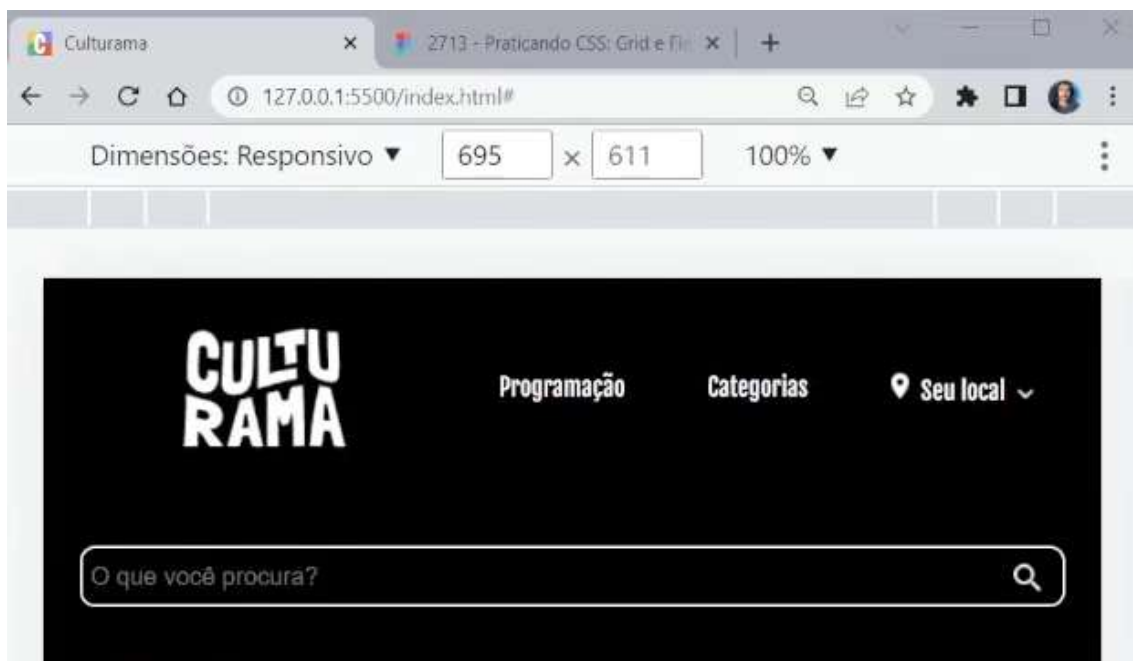
Ainda falta movermos a logo para o topo, deixando os três botões um ao lado do outro com o campo de pesquisa abaixo. Para isso, aplicaremos um espaçamento entre um item e outro com a propriedade `column-gap` recebendo o valor `33px`.

```
.menu {
  display: flex;
  justify-content: space-evenly;
  align-items: center;
  flex-wrap: wrap;
  column-gap: 33px;
}
```

Comparando com o projeto no Figma, é possível concluir que finalizamos o layout para celulares. Ao longo desse processo, já aprendemos um conjunto de propriedades muito recorrentes do Flexbox!

### Transcrição

Agora que já finalizamos o cabeçalho para celulares do projeto Culturama, vamos organizar os elementos para telas de tablets. Conferindo o projeto finalizado no Figma, vemos que é necessário apenas alinhar a logo e o botão "Seu local" com as margens da página. As imagens abaixo ainda não estão prontas, mas as margens sim (no arquivo `styles.css`), portanto podemos utilizá-las como referência.



Abrindo o arquivo `flex.css`, começaremos aplicando a instrução `@media`, que nos permitirá aplicar os estilos apenas para telas de pelo menos 720 pixels de largura. Dentro dos parênteses, passaremos o parâmetro `min-width: 720px`, abrindo e fechando chaves em seguida.

```
@media (min-width: 720px) {  
  
}
```

Vamos utilizar o "Inspecionar" do navegador, que nos auxiliará a construir esse layout. Conferindo o código que escrevemos, percebemos que já existe um `column-gap` de 33px aplicado. Na versão para tablets, podemos aumentar esse `column-gap` para expandir o espaçamento entre os itens, movendo a logo para a esquerda e o botão "Seu local" para a direita.

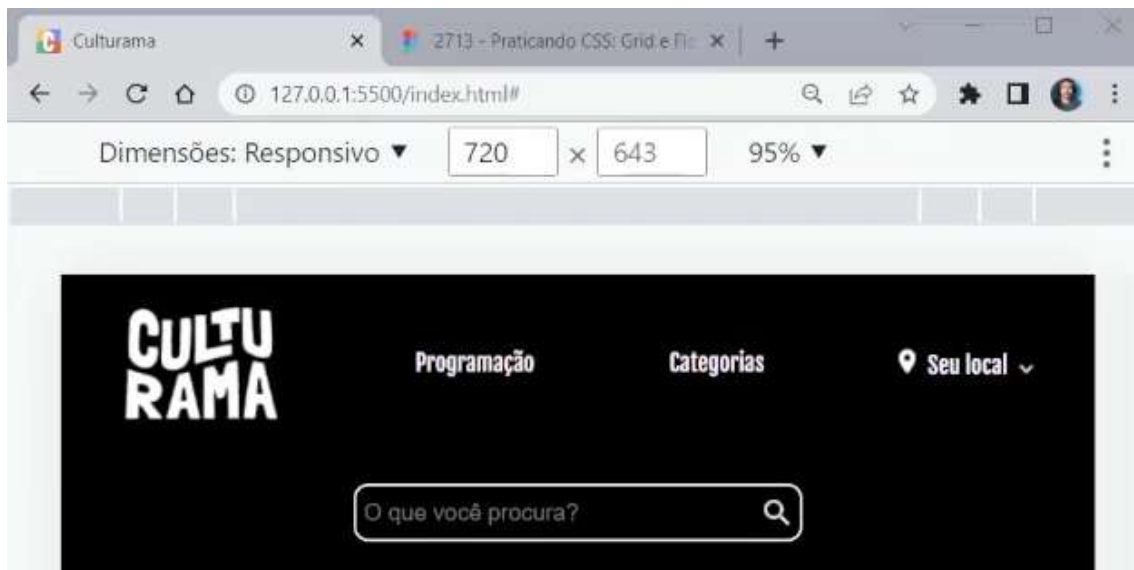
Fazendo alguns testes ainda no "Inspecionar", vemos que 75px é um valor adequado para alinhar os nossos elementos.

Não se esqueça de manter a janela do navegador em 720 pixels, medida de referência para telas de tablets!

No `@media` (), chamaremos a classe `.menu` e aplicaremos a propriedade `column-gap`, agora com 75 pixels.

```
@media (min-width: 720px) {  
  
  .menu {  
    column-gap: 75px;  
  }  
}
```

}



Comparando com o layout do Figma, é possível concluir que finalizamos o cabeçalho para telas de tablets. Além disso, aprendemos a utilizar o @media em conjunto com o Flexbox pra construir layouts responsivos.

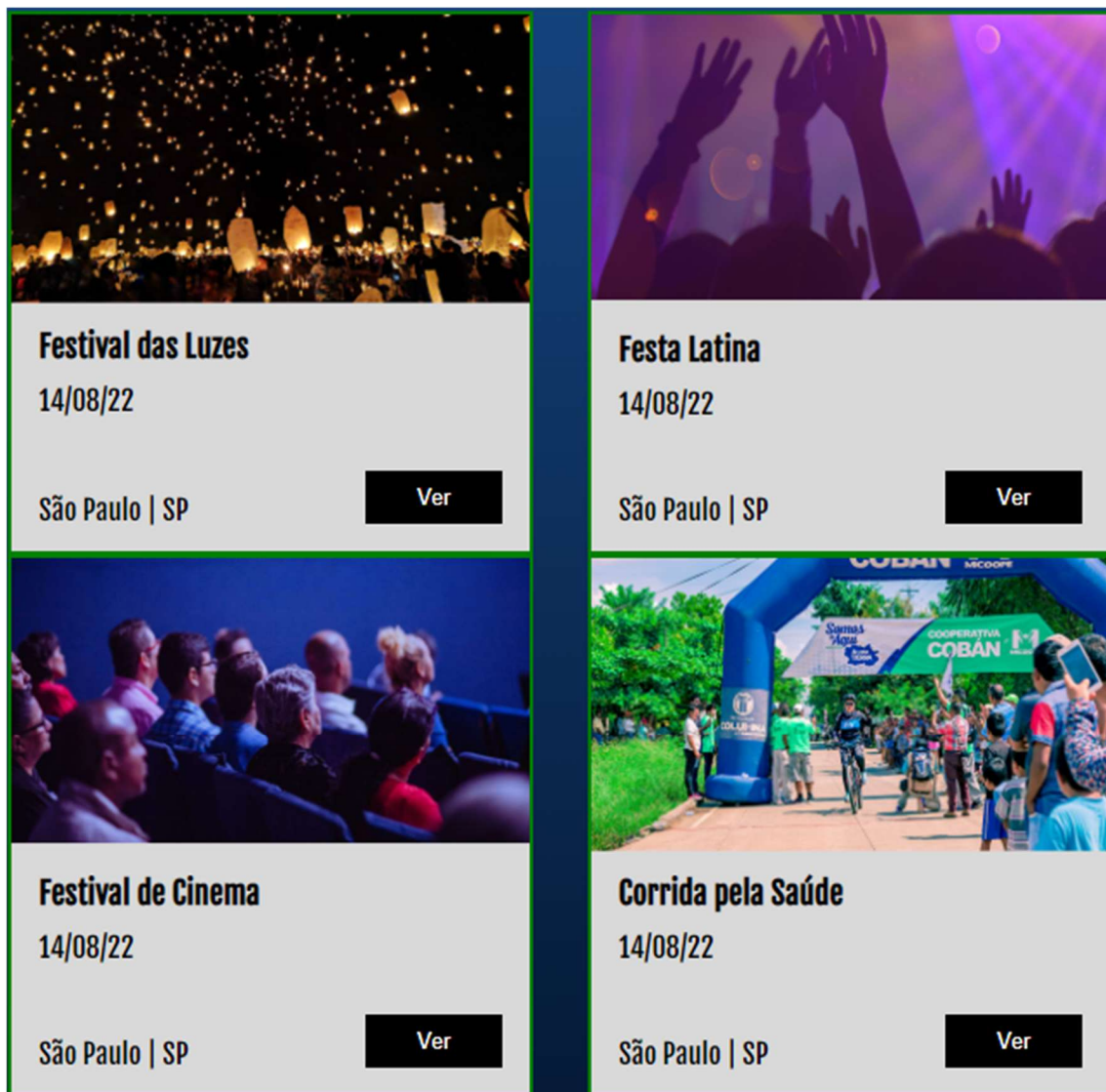
A propriedade gap do flexbox e grid, é utilizada para aplicar distância no sentido horizontal e vertical entre elementos em um layout, sua função é similar à propriedade margin.

### **column-gap**

A propriedade column-gap serve para criar uma lacuna vertical entre elementos.

Observe seu efeito na imagem a seguir:









### row-gap

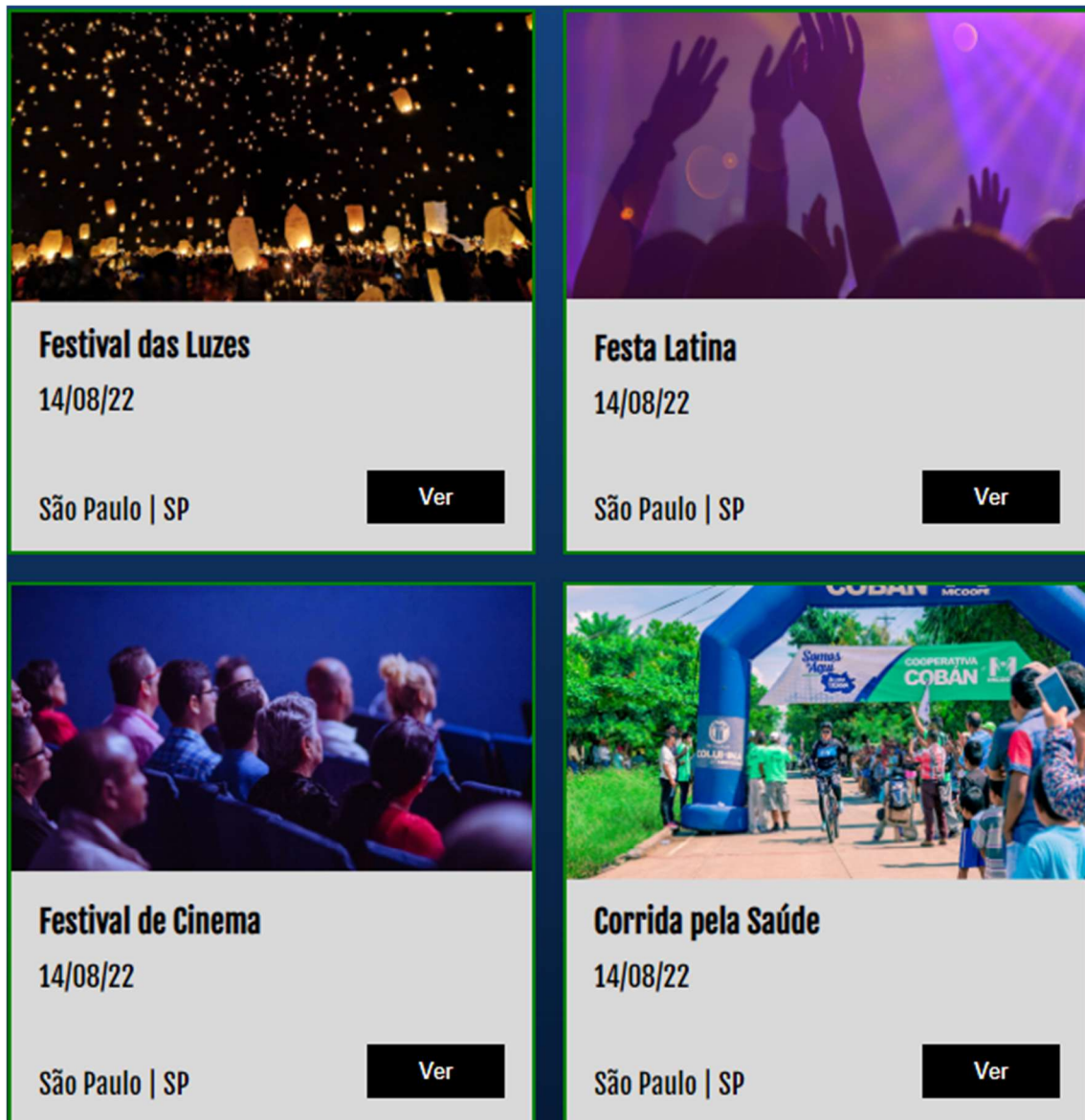
Já a propriedade `row-gap` serve para criar uma lacuna horizontal entre elementos.

Observe seu efeito na imagem a seguir:

 <p><b>Festival das Luzes</b> 14/08/22</p> <p>São Paulo   SP</p> <a href="#">Ver</a>	 <p><b>Festa Latina</b> 14/08/22</p> <p>São Paulo   SP</p> <a href="#">Ver</a>
 <p><b>Festival de Cinema</b> 14/08/22</p> <p>São Paulo   SP</p> <a href="#">Ver</a>	 <p><b>Corrida pela Saúde</b> 14/08/22</p> <p>São Paulo   SP</p> <a href="#">Ver</a>

### gap

E caso você utilize simplesmente a propriedade gap, ela irá criar espaçamentos tanto no sentido vertical quanto horizontal. Veja:



### Direção das Linhas e Colunas

Ao longo desse treinamento, você irá se deparar diversas vezes com a tarefa de organizar e dispor elementos na página e isso é feito durante o momento que deixamos claro a direção que estamos trabalhando.

Podemos citar que estamos trabalhando no eixo x ou no eixo y e uma outra maneira é dizer que estamos atuando no eixo horizontal ou eixo vertical.

### E por que é muito importante saber disso?

Porque **as direções são invertidas** no Flex e no Grid.

Bora lá entender!

### Trabalhando com Linhas e Colunas

No **flex**, quando utilizamos a propriedade flex-direction: row, estamos organizando os elementos um ao lado do outro, trabalhando com o eixo x ou horizontal.

Já no **grid**, quando utilizamos a propriedade `grid-template-rows` estamos calculando a **altura das linhas** e consequentemente, modificamos o eixo y ou vertical.

O mesmo ocorre com as colunas, no **flex** a propriedade `flex-direction: column` dispõe os elementos um abaixo do outro, ou seja, no eixo y ou vertical.

E no grid, a propriedade `grid-template-columns` serve para criar e calcular a **largura das colunas**, sendo assim, estamos alterando o eixo x ou horizontal.

Vamos finalizar o cabeçalho do projeto Culturama, agora trabalhando em telas de desktops.

Comparando o projeto em desenvolvimento com o resultado final esperado no modelo do Figma, vemos que a ordem dos elementos está errada: o campo de pesquisa deveria estar em segundo lugar, a logo deveria estar alinhada com a margem esquerda, e o último item deveria estar alinhado com a margem direita.

Não se esqueça de manter a janela do navegador em uma largura de 1440 pixels, referente a telas de desktops.

Voltando ao arquivo `flex.css`, usaremos novamente o `@media` para aplicar estilos em telas de, no mínimo, 1440 pixels. Criaremos uma nova instrução `@media` com esse valor para o parâmetro `min-width`.

```
@media (min-width: 1440px) {
```

```
}
```

Como é necessário alterar o espaçamento entre os itens, precisaremos ajustar o valor de `column-gap`. Chamaremos a classe `.menu` e aplicaremos essa propriedade com o valor 105px.

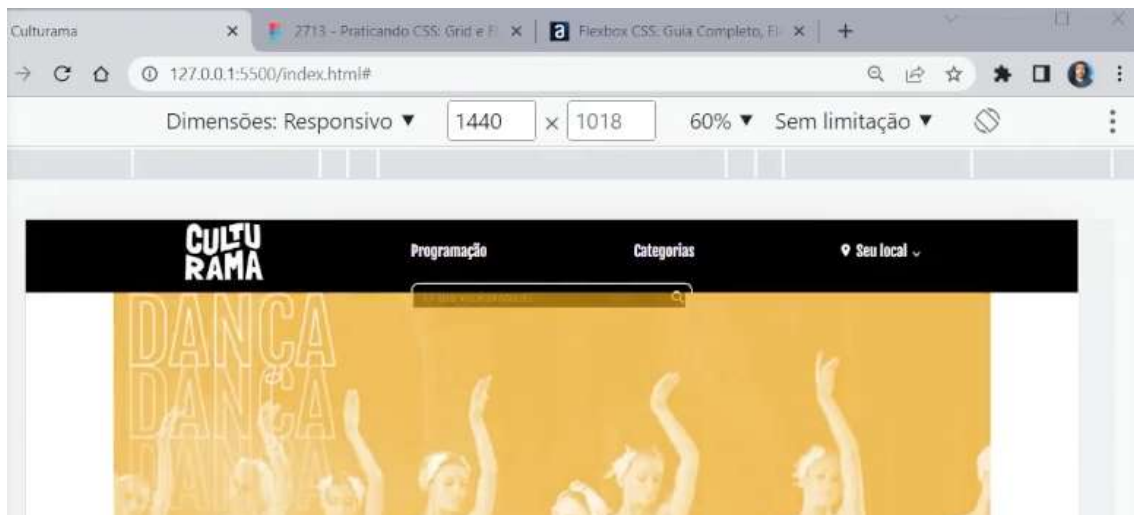
```
@media (min-width: 1440px) {
```

```
  .menu {
```

```
    column-gap: 105px;
```

```
  }
```

```
}
```



Porém, atualizando a página, veremos que o campo de pesquisa será movido para a linha abaixo, já que, devido ao aumento do espaçamento, existirá uma quebra de linha. Esse efeito é causado pela propriedade `flex-wrap`, que utilizamos anteriormente.

Para evitar isso, passaremos novamente essa propriedade, dessa vez com o valor `nowrap`. Assim, não teremos uma quebra de linha e todos os itens ficarão alinhados no mesmo nível.

**@media** (min-width: 1440px) {

  .menu {

    column-gap: 105px;

    flex-wrap: nowrap;

  }

}



O próximo passo será reordenar os elementos do nosso cabeçalho. No arquivo `index.html`, vemos que o campo de pesquisa se encontra em uma tag `<label>` com a classe `menu__label`. Poderíamos, por exemplo, mover essa tag para uma posição anterior - em segundo lugar na ordem dos elementos, e assim teríamos o resultado desejado. Entretanto, isso também alterará a ordem em telas menores, de tablets ou de celulares.

Pensando nisso, procuraremos outra solução, que também nos permitirá aprender mais sobre Flexbox. Para mudar a ordem de elementos, podemos usar a propriedade `order`.

Voltando ao `flex.css`, acessaremos o `@media` para telas superiores a 1440 pixels.

Adicionaremos a classe `.menu__item` e incluiremos uma pseudo classe usando a instrução `nth-child`, passando entre parênteses o valor 1.

```
@media (min-width: 1440px) {
```

```
  .menu {
    column-gap: 105px;
    flex-wrap: nowrap;
  }
```

```
  .menu__item:nth-child(1) {
  }
}
```

Dessa forma, conseguiremos aplicar estilos apenas para o primeiro elemento com a classe `menu__item`, nesse caso a logo. Para esse elemento, passaremos a propriedade `order: 1`, já que desejamos manter a sua posição.

```
@media (min-width: 1440px) {
```

```
  .menu {
    column-gap: 105px;
    flex-wrap: nowrap;
  }
```

```
  .menu__item:nth-child(1) {
    order: 1;
  }
}
```

Repetiremos o processo para o segundo item, o botão "Programação", que deve ficar em terceiro lugar; para o terceiro item, "Categorias", que deve ficar em quarto lugar; para o quarto item, "Seu local", que deve ficar em quinto lugar; e para o último item, o campo de pesquisa, que deve ficar em segundo lugar.

```
@media (min-width: 1440px) {
```

```
  .menu {  
    column-gap: 105px;  
    flex-wrap: nowrap;  
  }
```

```
  .menu__item:nth-child(1) {  
    order: 1;  
  }
```

```
  .menu__item:nth-child(2) {  
    order: 3;  
  }
```

```
  .menu__item:nth-child(3) {  
    order: 4;  
  }
```

```
  .menu__item:nth-child(4) {  
    order: 5;  
  }
```

```
  .menu__item:nth-child(5) {  
    order: 2;  
  }
```

```
}
```

Ao atualizar, todos os nossos itens estarão na ordem correta, sem que tenha sido necessário alterar a ordem do código no HTML.





Nós aplicamos a propriedade `order` diretamente a cada um dos itens que organizamos, e isso foi feito de forma diferente de todas as propriedades que usamos anteriormente. Isso porque estávamos trabalhando com *flex containers*, onde aplicamos uma propriedade ao elemento-pai de modo a organizar os elementos em seu interior.

Já a propriedade `order` foi aplicada nos filhos, ou seja, se caracteriza como uma propriedade de *flex items*.

Para aprender mais sobre as propriedades do Flexbox e seu comportamento, você pode acessar o [artigo sobre Flexbox CSS da Juliana Amoase!](#)

Agora que finalizamos o cabeçalho do nosso projeto para as três telas, passaremos para o banner na parte inferior. Analisando visualmente, já é possível perceber que será difícil trabalhar essas imagens com o Flexbox.

Isso porque a primeira e a última imagem ocupam duas linhas, enquanto as imagens centrais ocupam uma linha cada uma. Tal comportamento poderia ser feito com o Flexbox se estivéssemos trabalhando com telas estáticas, mas alterar a largura da tela vai quebrar os elementos.

Você pode ler o artigo [“Flexbox CSS: Guia Completo, Elementos e Exemplos”](#) escrito pela instrutora Juliana Moasei e conhecer todas as propriedades dos tipos `flex-item` e `flex-container` do Flexbox.

## Flex

Chegou o momento de você colocar em prática um pouco do que aprendeu nesta aula, existem diversas técnicas para centralizar elementos na tela, mas nesse desafio o propósito é que você centralize utilizando apenas propriedades flex.

A base do código HTML é esta:

```
<body>
```

```
  <main>
```

```
    <div>
```

```
      <p>Me centraliza!</p>
```



```
    </div>
  </main>
</body>
```

E a base de código CSS é essa:

```
body {
  background-image: url("https://cdn3.gnarususercontent.com.br/alura%2B/bg---alura-.png");
  background-repeat: no-repeat;
  background-size: cover;
  font-family: 'Fjalla One', sans-serif;
}

main {
  height: 100vh;
  width: 100vw;
}

div {
  width: 240px;
  height: 110px;
  font-size: 36px;
  font-weight: bold;
  background-color: #fff;
  border-radius: 4px;
  padding: 0 10px;
  background-image: linear-gradient(315deg, #4dccc6 0%, #96e4df 74%);
  box-shadow: 4px 4px 6px 0 rgba(255, 255, 255, .3),
    -4px -4px 6px 0 rgba(116, 125, 136, .2),
    inset -4px -4px 6px 0 rgba(255, 255, 255, .2),
    inset 4px 4px 6px 0 rgba(0, 0, 0, .2);
}
```

No momento, o resultado no navegador está dessa forma:



**Me centraliza!**

E assim é como tem que ficar o resultado final:



**Me centraliza!**

**Dica:** O grande objetivo é que essa div fique no centro da página, tanto verticalmente quanto horizontalmente. Para isso, utilize propriedades que alinham no eixo principal e no eixo transversal.

A propriedade `justify-content` alinha itens no eixo principal, e a propriedade `align-items` alinha itens no eixo transversal. Aplicando o valor `center` para ambas propriedades, o item ficará no centro do seu container.

Código CSS:

\\\ código acima ocultado

```
main {
```

```

height: 100vh;

width: 100vw;

display: flex;

justify-content: center;

align-items: center;

}

```

\\\ código abaixo ocultado

Parabéns! div centralizada com sucesso.

Eu estou com o Figma do Culturama aberto na tela de 360 pixels. Nessa aula, daremos play no grid e começaremos o nosso trabalho. Iniciaremos pelo *grid container banner*, que tem duas colunas e três linhas.

Ele também aloca quatro elementos que atuam como *grid items*: a imagem amarela que ocupa a primeira linha e as duas colunas; a imagem lilás, ocupando a primeira coluna e a segunda linha; a imagem verde, que ocupa a terceira linha e a primeira coluna; e, por fim, a imagem em vermelho, ocupando a segunda coluna, além da segunda e terceira linhas. Além disso, existe um espaçamento (ou *gap*) entre as linhas e as colunas.

Abrindo o nosso navegador, podemos reparar que os elementos filhos de banner se encontram um abaixo do outro. Qual a nossa tarefa, nesse caso? Primeiro, precisamos trazer o contexto do *display grid* para esse *container*, adicionar os espaçamentos e posicionar cada *grid item* em seu devido lugar conforme o Figma.

Vamos ao VS Code? Nele, criaremos um arquivo chamado *grid.css* na pasta *style*. Depois, copiaremos a linha 14 do HTML, colaremos na linha de cima e substituiremos o flex por grid, resultando na seguinte linha atualizada:

```
<link rel="stylesheet" href="./assets/style/grid.css">
```

Em seguida, salvaremos o arquivo.

No arquivo *grid.css*, chamaremos esse elemento correspondente à seção do banner. Sua classe é banner:

```

.banner {

  display: grid;

  gap: 1rem;

}

```

Depois disso, salvamos as alterações. Podemos perceber que o espaçamento foi aplicado, mas ele não gerou grandes modificações do ponto de vista visual.

No entanto, surgiu um botão chamado grid no elemento "banner". Esse botão significa que o banner agora faz parte de um contexto de grid e cada um de seus elementos filhos está alocado em sua respectiva célula.

Por mais que o navegador seja muito esperto, ele não consegue adivinhar onde cada um desses itens deve se posicionar. Por isso, conto com você para me ajudar a passar essa informação a ele.

Um elemento passa a ser interpretado como um **grid container**, a partir do momento em que é aplicado nele a propriedade `display:grid`, a partir disso, cada elemento filho direto passa a ser visto como um **grid item**.

Esses termos são de fundamental importância pois existem propriedades que obrigatoriamente devem ser aplicadas ao elemento pai (Grid Container) e outras que devem ser aplicadas exclusivamente ao elemento filho (Grid Item).

Muitas delas serão vistas ao longo desse treinamento!

Grid container	Grid item
<code>display: grid</code>	<code>grid-column</code>
<code>grid-template-areas</code>	<code>grid-area</code>
<code>column-gap, gap, row-gap</code>	<code>align-self</code>
<code>grid-template-rows</code>	<code>grid-row</code>
<code>grid-template-column</code>	

Além disso, caso você queira consultar as propriedades restantes, acesse esse [Guia do CSS tricks](#), porém o conteúdo se encontra em inglês.

Estou com o navegador aberto, onde o banner já está sendo interpretado como um grid container. O que precisamos fazer agora? Posicionar cada grid item em seu devido lugar conforme as linhas e colunas do container. Vamos lá!

Abaixo do bloco do banner no VS Code, escrevemos o seguinte:

```
.banner img:nth-child(1) {
```

```
grid-column: 1/3;
}
```

Ao inserirmos o número 1 entre os parênteses do `nth-child()`, conseguimos pegar o primeiro elemento filho de banner que seja uma imagem. Após o fechamento desse bloco, escrevemos:

```
.banner img:nth-child(4) {
  grid-column: 2/3;
  grid-row: 2/4;
}
```

Quando inserimos o número 4 entre os parênteses do `nth-child()`, pegamos o último elemento filho de banner que seja uma imagem, ou seja, a imagem vermelha.

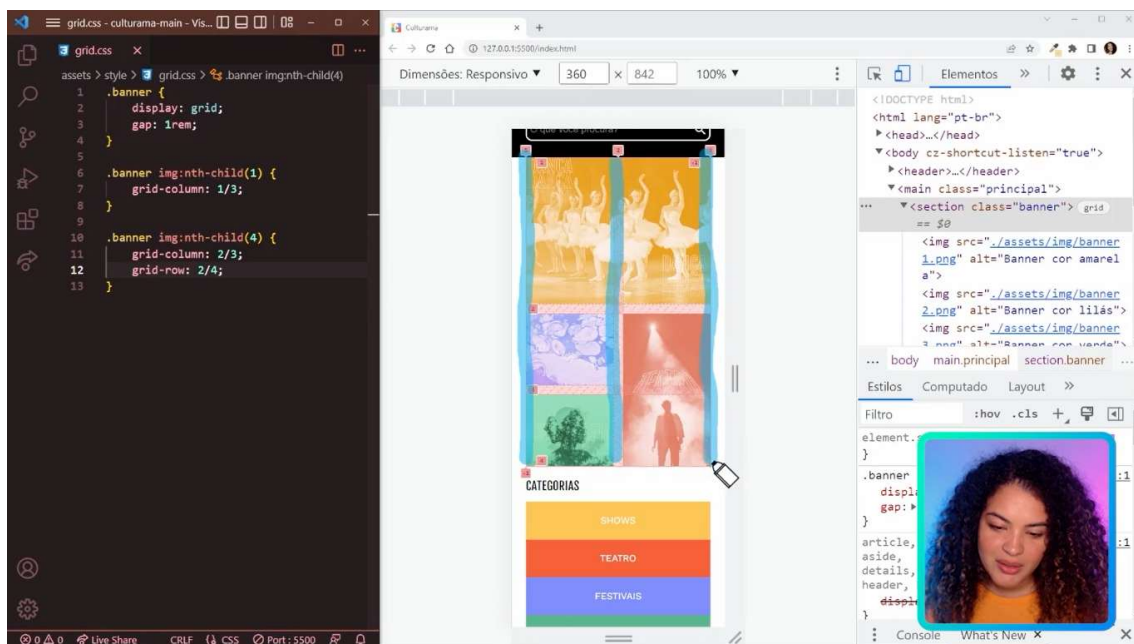
Após salvar, veremos que os itens estão posicionados em seu devido lugar. Repare que bastou posicionar o primeiro e o último itens que o restante já foi posicionado automaticamente.

Agora, vamos entender o que acabamos de construir: as propriedades `grid-column` e `grid-row` determinam a posição inicial e final que o elemento irá ocupar.

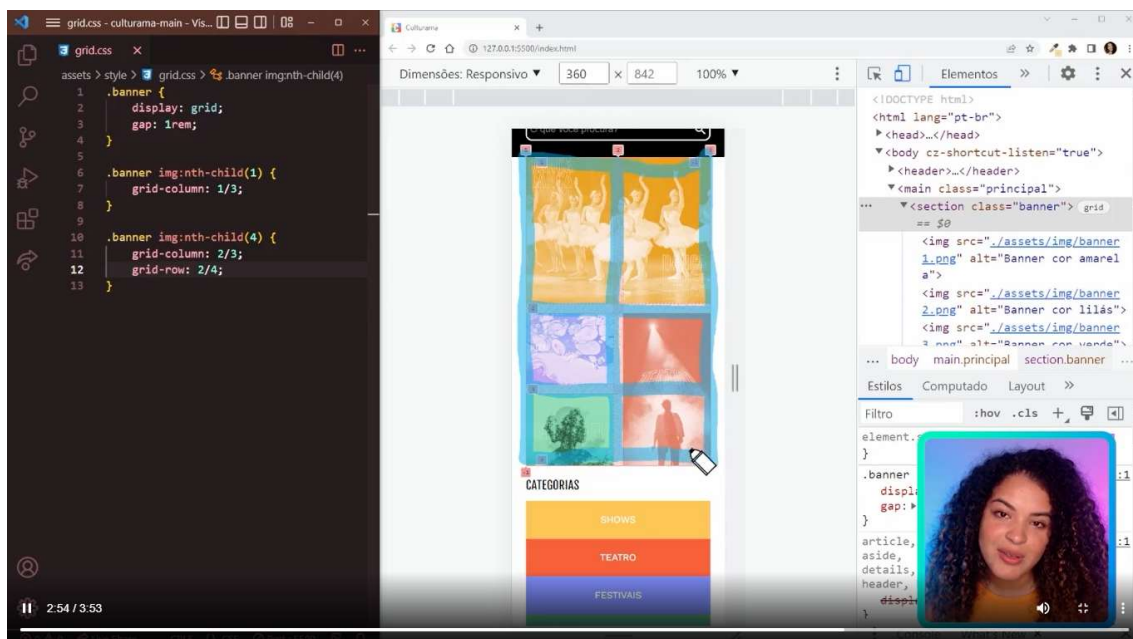
Por isso, elas recebem dois valores separados por uma barra. O primeiro valor se refere à posição inicial e o segundo, à posição final.

Eu sei que você deve estar se perguntando: "mas como eu vou saber qual valor colocar? Devo tirar isso da minha cabeça?". Não! Você saberá por meio da contagem dos traços do grid, ou seja, as *grid lines*. Vamos contar e ver como isso funciona?

Quantas colunas o banner tem? Duas. E quantos traços existem em duas colunas? São três traços, sendo dois na lateral inicial e final de cada coluna e uma exatamente entre elas.



Isso também serve para as linhas: se temos três linhas, teremos quatro traços, um acima da primeira linha e outro abaixo da última, além de dois intermediários, um entre a segunda e a terceira linha e outro entre a terceira e a quarta linha.



O que isso quer dizer? Significa que, se queremos que a imagem amarela ocupe a primeira e a segunda coluna, precisamos indicar que seu `grid-column` é de `1/3`, ou seja, ela ocupa o espaço entre o traço 1 e o traço 3 no que diz respeito às colunas.

Já se queremos que a imagem vermelha ocupe a linha 2 e a linha 3, precisamos indicar que seu `grid-row` é de `2/4`, ou seja, ela ocupa o espaço entre o traço dois e o traço 4.

Posicionamos os grid items para a tela de 360 pixels, mas abrindo o Figma, percebemos que nessa resolução o banner possui sua própria disposição entre linhas e colunas. A partir de 720 pixels, essa disposição mudará, mudando também o posicionamento dos itens. Como podemos solucionar isso?

Note que o posicionamento dos itens no Figma foi modificado a partir da tela de 720 pixels. Isso porque os valores se invertem.

Na tela de **360 pixels**, o banner possui **três linhas e duas colunas**. Já na tela de **720 pixels**, ele passa a ter **duas linhas e três colunas**. Consequentemente, tudo muda.

A imagem amarela, por exemplo, ocupava a primeira linha, além da primeira e a segunda colunas na tela de 360 pixels. Já na tela de 720 pixels, ela passa a ocupar apenas a primeira coluna, além da primeira e segunda linhas.

Se eu aumentar a tela até 720 pixels, repare que a modificação no posicionamento não foi reconhecida. Isso porque nós não passamos essa informação para o navegador. Esse será o nosso trabalho agora. Vamos lá!

No VS Code, escreveremos:

**@media** (min-width): 720px) {

```

.banner img:nth-child(1) {
  grid-column: 1/2;

```

```

    grid-row: 1/3;
}

.banner img:nth-child(4) {
    grid-column: 3/4;
    grid-row: 1/3;
}
}

```

Repare que copiamos o conteúdo entre a linha 6 e a linha 13, parte que tem as informações de posição das imagens no grid container, e modificamos o grid-column do banner img 1 para 1/2, além de acrescentar um grid-row de 1/3. Fizemos o mesmo com o banner img 4 para colocá-lo em uma posição de grid-column de 3/4 e grid-row de 1/3.

Após salvar, nossos elementos já estão na posição certa para as telas acima de 720 pixels. O resultado é que a imagem amarela ocupa a primeira coluna, na primeira e segunda linhas, e a imagem vermelha ocupa a terceira coluna, também na primeira e segunda linhas.

Com isso, o banner já está quase finalizado. Partirmos para o próximo container.

### **Mapa do Grid do Banner**

Agora que você aprendeu sobre a contagem das linhas do Grid, o posicionamento de cada grid item e a disposição entre linhas e colunas, repare que todas essas informações em conjunto acabam formando um mapa do grid do elemento, que foi desenhado mentalmente durante os vídeos anteriores.

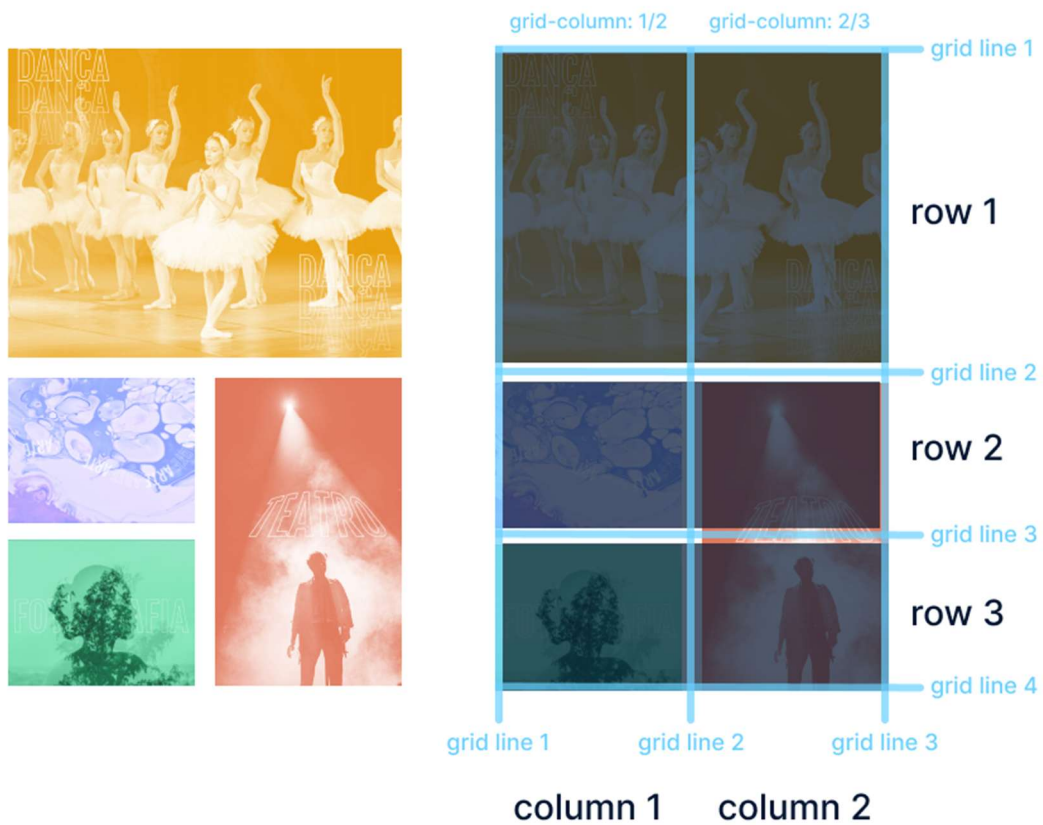
Para visualizar tudo de forma muito mais nítida, coloquei abaixo um mapa do grid do banner com os elementos que você acabou de posicionar para criar um layout.

### **Tela de 360px**



## Mapa do Grid

Tela 360px | Linhas e colunas



grid-row: 1/2  
grid-column: 1/3

grid-row: 2/3  
grid-column: 1/2

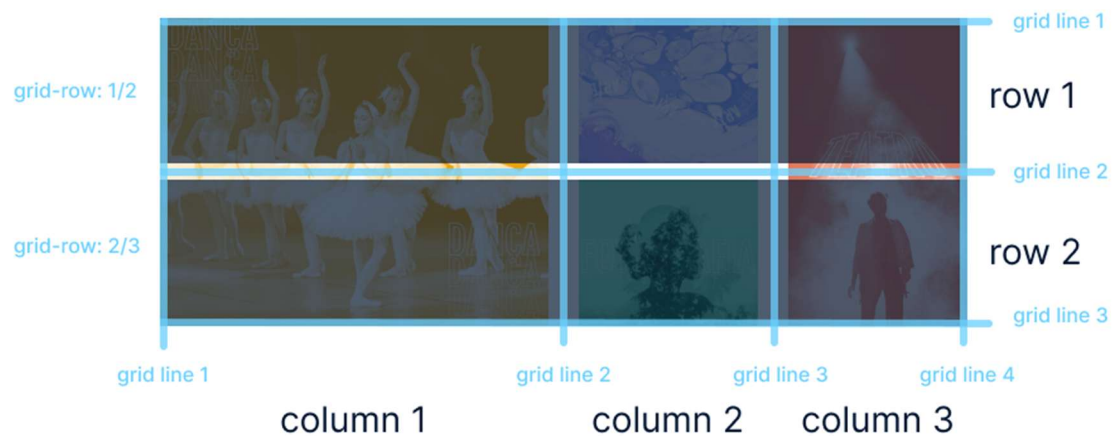
grid-row: 3/4  
grid-column: 1/2

grid-row: 2/4  
grid-column: 2/3

Tela de 720px

## Mapa do Grid

Tela 720px | Linhas e colunas



grid-row: 1/3  
grid-column: 1/2

grid-row: 1/2  
grid-column: 2/3

grid-row: 2/3  
grid-column: 2/3

grid-row: 1/3  
grid-column: 3/4

Esse modelo é para salvar no coração! Pois sempre que quiser criar um grid layout para um elemento, essa é uma estratégia que você pode utilizar como aliada, ou seja, desenhar um mapa mental de como você gostaria que aquele Grid Container se apresente.

### Detalhando com start e end

Você sabia que as propriedades grid-column e grid-row são abreviações de outras duas propriedades?

Pois é, veja no exemplo do banner!



Supondo que a imagem amarela tenha uma classe chamada “imagem-amarela”, o código que definirá o seu posicionamento é:

```
.imagem-amarela {
  grid-column: 1/2;
  grid-row: 1/3;
}
```

E podemos obter o mesmo resultado desse código, de uma maneira mais detalhada, utilizando o start e o end, veja:

```
.imagem-amarela {
  grid-column-start: 1;
  grid-column-end: 2;
  grid-row-start: 1;
  grid-row-end: 3;
}
```

O start irá pegar o primeiro valor do grid-column e grid-row e o end receberá o segundo valor dessas propriedades.

### Compactando com grid-area

O grid-column e o grid-row possuem informações compiladas, mas é possível condensar ainda mais, com o uso da propriedade grid-area.

O código fica assim:

```
.imagem-amarela {
  grid-area: 1 / 1 / 3 / 2
}
```

A propriedade `grid-area` irá admitir os 4 valores relacionados ao `grid-column` e `grid-row`, mas um ponto muito importante é respeitar a *ordem* desses valores, que pode ser visualizada na ferramenta do desenvolvedor.

```

☒ grid-area: 1 / 1 / 3 / 2;
    grid-row-start: 1;
    grid-column-start: 1;
    grid-row-end: 3;
    grid-column-end: 2;

```

### Uma última dica

No início, pode ser difícil compreender como funciona a contagem das grid lines mas existe uma regrinha que pode nos ajudar bastante, veja nos exemplos abaixo.

**Exemplo 1:** A imagem amarela ocupa a linha 1 e a coluna 1, então o seu `grid-row-end` e `grid-column-end` terá em ambos o valor 2.

**Exemplo 2:** A imagem vermelha ocupa a coluna 3, então o seu `grid-column-end` terá o valor 4. Além disso, essa imagem ocupa as linhas 1 e 2, com isso o `grid-row-end` será 3.

Sendo assim, temos o seguinte modelo:

O valor do end será sempre **um número a mais** que o número das linhas e colunas que o elemento ocupa.

Seja muito bem-vindo e muito bem-vinda à terceira aula do Projeto Praticando CSS: Grid e Flexbox. Nessa aula, organizaremos as áreas das seções de categorias, destaques e próximos eventos.

Começarei clicando com o botão direito na página e selecionando "Inspecionar" no navegador. Em seguida, aplicarei a largura de 360 pixels, usada para dispositivos móveis, já que estamos fazendo o *Mobile First*, ou seja, "primeiro para celulares".

Começaremos organizando a parte de categorias. Além disso, conferiremos como está a área de categorias no Figma para saber qual o nosso objetivo.

Fazendo essa comparação com o Figma, percebemos que é preciso apenas aplicar um espaçamento entre os elementos. Para fazer isso, dividirei a tela do computador, deixando o navegador do lado direito e o VS Code do lado esquerdo.

Agora, inseriremos alguns códigos flexbox no arquivo `flex.css`. Nessa etapa do projeto, já temos disponível o arquivo `grid.css`.

Usarei o Inspecionar para conferir como está a seção atual de categorias. Encontraremos essa seção descrita da seguinte forma:

```
<section class="categorias">...</section>
```

Podemos ver que a seção contém um `display: block;` que deixa todos os elementos uns em cima dos outros, ocupando toda a largura disponível. Queremos alterar esse `display` para `flex`, já que o nosso objetivo é organizar os elementos com o Flexbox.

Para isso, começaremos procurando os elementos da lista de categorias no código HTML. Eles se encontram da linha 49 até a 56, dentro da *section* "categorias":

```
<section class="categorias">

  <h2 class="categorias__titulo">Categorias</h2>

  <h1 class="categorias__lista">

    <li class="categorias__item">Shows</li>

    <li class="categorias__item">Teatro</li>

    <li class="categorias__item">Festivais</li>

    <li class="categorias__item">Cinema</li>

    <li class="categorias__item">Arte</li>

    <li class="categorias__item">Fotografia</li>

    <li class="categorias__item">Tecnologia</li>

    <li class="categorias__item">Design</li>

  </ul>
```

Para organizar esses elementos com o Flexbox, precisamos localizar o seu pai, a propriedade `<h1 class="categorias__lista">`, na linha 48.

Levaremos essa classe para o nosso arquivo `flex.css`. Escreveremos o seguinte na linha 9:

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

Ao atualizar a página, veremos que os elementos ficam todos em linha horizontal, um ao lado do outro. Esse é o comportamento padrão do `display: flex`. Queremos alterar o eixo de uma linha horizontal para uma coluna vertical.

Como fazemos isso com o Flexbox? Podemos alterar o eixo com a propriedade `flex-direction`, que aponta a direção do flex. Como queremos uma coluna, aplicamos o valor `column` a essa propriedade, resultando no seguinte código:

```
.categorias__lista {
  display: flex;

  flex-direction: column;
}
```

Ao atualizarmos, veremos que os elementos ficaram novamente alinhados na vertical, um em cima do outro, parecido com a disposição do `display: block`.

Ficou faltando, então, aplicar o espaçamento entre os elementos. Conferindo o nosso projeto no Figma, percebemos que esse espaçamento está na horizontal, ou seja, corresponde a uma linha entre as seções.

Podemos aplicar esse espaçamento com a propriedade `row-gap`, que permite inserir um intervalo entre as linhas. O valor usado será de `.5rem`, resultando no seguinte código:

```
.categorias__lista {  
  display: flex;  
  flex-direction: column;  
  row-gap: .5rem;  
}
```

Com isso, finalizamos a nossa edição da seção `categorias` para celulares. Além disso, você aprendeu a alterar o eixo de elementos de linha para coluna. Nos próximos vídeos, praticaremos e aprenderemos mais sobre o Flexbox.

Continuaremos a trabalhar na seção de categorias do projeto Culturama. Primeiro, analisaremos o seguinte comportamento: se transformarmos todo o código que fizemos no vídeo anterior em comentário e atualizarmos o arquivo `flex.css`, veremos que os elementos ficam uns em cima dos outros, parecido com o nosso objetivo final.

Não sei se você chegou a se perguntar o porquê de usar o `display: flex` se o `display: block` já nos proporciona um resultado semelhante. Ao usar essa segunda propriedade, bastaria aplicar um `margin-bottom` entre os elementos e a seção ficaria pronta.

Porém, quando analisamos a seção `categorias` para as telas de tablets no layout do Figma, vemos que as categorias são divididas entre duas colunas, diferentemente do que acontece no mobile, que tem apenas uma coluna.

Essa configuração não é possível de ser alcançada com o `display: block`, pois este permite apenas um elemento por linha, ocupando toda a largura do layout.

Já o `display: flex` é mais flexível e permite que quebrems uma coluna em várias ou uma linha em muitas. Esse é o nosso desafio para este vídeo: dividir a coluna única do mobile em duas para podermos finalizar a seção de categorias para tablets.

Começarei retirando a anotação de comentário do código da linha 9 à linha 13 e aumentando a largura do layout para 720 pixels no Inspeccionar. Lembrando que essa é a largura padrão para telas de tablets.

Como vou inserir estilos apenas quando a tela estiver maior, vou aplicar novamente a classe na linha 21, dentro do `@media` para as telas de tablet.

Como queremos que haja uma quebra de coluna, usaremos a mesma propriedade usada anteriormente no projeto, a `flex-wrap` com o valor `wrap`. Ao atualizar, veremos que a coluna única foi quebrada em duas no tablet.

Agora, precisamos aplicar um espaçamento entre as duas colunas. Para isso, podemos usar a propriedade `column-gap` para criar um intervalo entre as colunas. O valor desse gap será de `1.5rem`. O resultado será o seguinte código:

```
.categorias__lista {
  flex-wrap: wrap;
  column-gap: 1.5rem;
}
```

Ao atualizar a página, veremos que as categorias ficaram bem alinhadas, inclusive com o banner.

O próximo passo é aumentar o espaçamento entre as linhas dos elementos. Então, usaremos a propriedade `row-gap` com um valor de `1rem`.

```
.categorias__lista {
  flex-wrap: wrap;
  column-gap: 1.5rem;
  row-gap: 1rem;
}
```

Após a atualização, percebemos que o resultado está similar ao projeto no Figma. Com isso, finalizamos a seção categorias para as telas de tablets.

Nessa etapa do projeto, já estamos fazendo os códigos da parte de menu e de categorias. Para o nosso código ficar mais organizado e legível, começaremos a inserir comentários.

Na linha 1, comentaremos que essa parte do código corresponde ao cabeçalho.

Selecionaremos o texto e pressionaremos o atalho "CTRL + ;" para inserir o comentário, como o seguinte:

```
/* Cabeçalho */
```

Faremos o mesmo com a parte de menu dentro do `@media` e a parte de categorias tanto na linha 11 quanto no `@media`. Com isso, finalizamos a seção de categorias para as telas de tablet. A seguir, praticaremos e aprenderemos mais conceitos de Flexbox.

Agora, conferiremos a parte abaixo da seção de categorias (retomaremos esta seção depois).

Focando na parte de destaques, vejo que os elementos estão um em cima do outro porque estão com o `display: block`. Eu preciso alterar esse `display` para `flex`, para que eles fiquem na mesma linha, um ao lado do outro.

Procurarei esses elementos no nosso código HTML. Eles estão entre as linhas 59 e 66:

```
<div class="categorias__destaques">
  <div class="destaques">
    <h3 class="destaques__titulo">Destaques:</h3>
    <div class="destaques__barra"></div>
  <div>
    
```

```

        
    </div>
</div>
</div>
<div class="categorias__imagem"></div>

```

Na linha 60, temos o título Destaques, na linha seguinte, temos uma div com uma classe chamada destaques\_\_barra. A linha horizontal preta que vemos logo abaixo do título no layout está aplicada com um background nessa classe.

Temos também uma div da linha 62 à linha 65, dentro da qual temos os ícones de setas para a esquerda e para a direita. Para organizar esses elementos com o Flexbox, precisamos localizar seu pai, que corresponde à div da linha 59 com a classe destaques.

Levaremos essa classe para o nosso arquivo flex.css. Na linha 19, escreveremos .destaques { e já aplicaremos o display: flex. Com isso, poderemos observar que os itens já ficam todos na mesma linha.

Porém, quando aplicamos o display: flex, os itens ficam todos alinhados ao topo da linha. Para podermos organizá-los verticalmente, podemos usar a propriedade align-items com o valor center, para centralizá-los na vertical. Com isso eles ficam todos no mesmo nível.

Também vamos inserir um gap de 10 pixels para que eles fiquem um pouco distantes uns dos outros. O gap é aplicado tanto no sentido das linhas quanto das colunas, mas, no nosso caso, como não há um elemento acima do outro, o gap não será aplicado em linha, apenas em coluna.

Outra alternativa seria usar o column-gap com o mesmo intervalo de 10px. O resultado será o mesmo.

O código ficou assim:

```

.destaques {
    display: flex;
    align-items: center;
    column-gap: 10px;
}

```

Agora, testaremos a responsividade do nosso projeto para verificar se ele se ajusta a diferentes tamanhos de tela. Quando mudamos a largura do nosso layout com o Inspeccionar, vemos que os elementos não estão se ajustando conforme a largura do layout.

Precisamos, por exemplo, que as setas para a direita e para a esquerda se expandam e ocupem toda a largura disponível para esses elementos.

Conseguimos esse resultado da seguinte maneira: escrevemos na linha 25 a classe .destaques\_\_barra e aplicamos a propriedade flex-grow com o valor 1.



Após atualizar, se fizermos um novo teste da responsividade do projeto, a barra crescerá. Também podemos fazer com que os ícones não quebrem (ou seja, apareçam um sobre o outro a depender do tamanho da tela) aplicando um white-space de valor nowrap no Inspeccionar.

Com essas modificações, a barra ocupa toda a largura do container. O flex-grow é uma propriedade de flex-item. Repare que a apliquei diretamente no elemento que eu desejava modificar na barra em vez de aplicá-la ao container ou elemento pai.

Após esse vídeo, deixarei algumas atividades para você exercitar ainda mais a propriedade flex-grow.

A propriedade flex-grow é utilizada para expandir elementos de acordo com o espaço disponível em um flex-container. Porém, como é possível calcular essa proporção?

Vamos conferir nos exemplos abaixo:

#### **Aplicando flex-grow com o mesmo valor para todos os elementos**

Supondo que o container tenha 1000px de largura, e 4 cards com 200px cada, sobrando 200px de espaço livre. Esse espaço livre será dividido igualmente e somado à largura de cada item, dando o total de 250px para cada item.

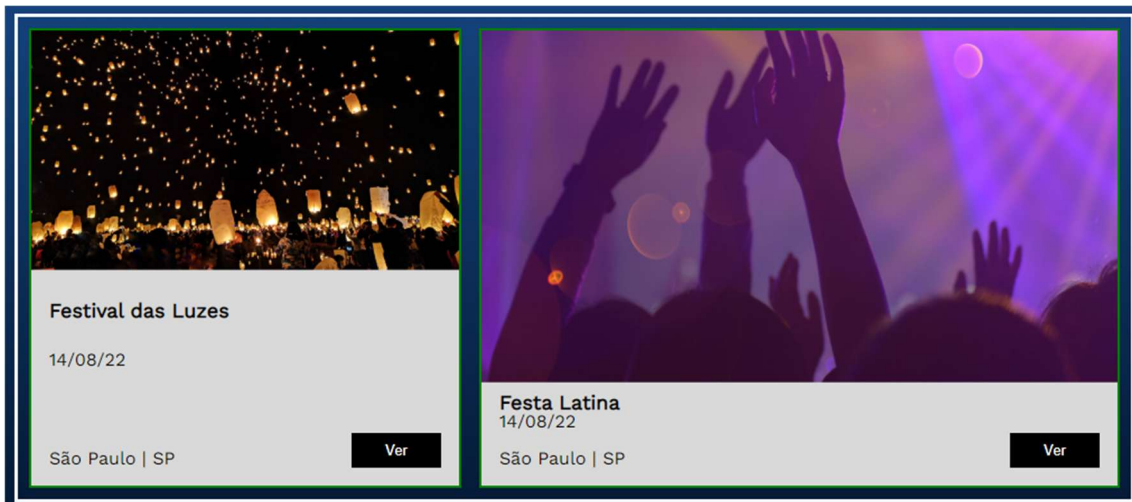


#### **Aplicando flex-grow com valores diferentes para cada elemento**

Supondo que o container tenha 1000px de largura, e 2 cards com 200px de largura, sobrando 600px de espaço livre.

Aplicando no primeiro card flex-grow: 1, e no segundo card flex-grow: 2. O segundo card irá crescer proporcionalmente o dobro que o card 1, ficando com mais 400px, dando o total de 600px de width. Já o primeiro card terá mais 200px, dando o total de 400px de width.

**Importante destacar que o elemento não fica com o dobro de largura, porém cresceu proporcionalmente o dobro**



**Obs: valores de gap, border e, padding podem influenciar no comportamento do flex-grow.**

Já trabalhamos bastante nas seções categorias e destaques do nosso projeto Culturama. Agora, partiremos para a seção seguinte, a de próximos eventos.

Nela, temos uma lista de elementos que estão todos um acima do outro, pois sofrem a influência do display: block que não nos permite deixar um elemento ao lado do outro. Percebemos então que precisamos alterar essa disposição para o display: flex.

Procurarei esses elementos no nosso código HTML, que começa na linha 73 e vai até a linha 131 do nosso arquivo. Para aplicar o display: flex a esses elementos, preciso localizar o seu pai, que é uma ul na linha 73 com a classe eventos\_\_lista:

```
<section class="eventos">

  <h2 class="eventos__titulo">Próximos eventos</h2>

  <ul class="eventos__lista">
```

Levaremos essa classe para o nosso arquivo flex.css. Escreveremos .eventos\_\_lista na linha 29 e aplicaremos o display: flex. Ao atualizar, perceberemos que os elementos são forçados a ficar na mesma linha. Para evitar que isso aconteça, usaremos um flex-wrap com valor wrap.

Na linha 19, inseriremos o comentário /\* Destaques \*/ para deixar o código mais organizado. Faremos o mesmo logo acima da classe eventos, inserindo /\* Eventos \*/ na linha 31.

Agora, está faltando aplicarmos um espaçamento entre as colunas. Faremos isso inserindo um column-gap com valor de 1.5rem, mesmo gap que utilizamos no restante do projeto.

O próximo passo é aplicar um espaçamento na horizontal entre os itens da lista. Para isso, colocaremos um row-gap de 1rem. O resultado é o seguinte:

```
.eventos__lista {
  display: flex;

  flex-wrap: wrap;

  column-gap: 1.5rem;

  row-gap: 1rem;
```

```
}
```

Em seguida, testaremos a responsividade do projeto, ou seja, como ele se adapta a diferentes tamanhos de tela.

Ao fazermos o teste, percebemos que surge um espaço em branco na lateral direita do layout para uma largura de 966 pixels. Para solucionar esse problema, localizaremos a classe `__item` na linha 75, ou seja, `eventos__item`, e a levaremos para o arquivo `flex.css`.

Em seguida, aplicaremos um `flex-grow` de valor 1. Ao atualizar, faremos um novo teste no Inspeccionar. Quando aumentamos a largura do layout, os itens vão se ajustando e ocupando toda a largura do container, mas esse ajuste está um tanto descontrolado.

Seria bom inserirmos um limite de largura para os itens não se expandirem demais, até para facilitar a organização dos elementos em cada li. Para fazer isso, inseriremos um `max-width` de 400px:

```
.eventos__item {
  flex-grow: 1;
  max-width: 400px;
}
```

Ao testar novamente, percebemos que os elementos não crescem mais desproporcionalmente: eles aumentam até o limite de 400 pixels de largura e param de crescer.

A propriedade `flex-grow` foi usada de três formas diferentes ao longo do Projeto: primeiro, ela foi usada na barra preta ao lado do título destaques e antes disso aprendemos a usar o `flex-grow` em itens irmãos com valores diferentes e vimos como isso influencia a sua distribuição no layout.

Nesse vídeo, usamos a propriedade com o mesmo valor para todos os itens, informando que queremos que todos eles cresçam proporcionalmente iguais.

Agora, faremos uma comparação entre o layout mobile no Figma com o Inspeccionar para verificar o que falta ser feito na largura de 360 pixels. Analisando o final do layout, percebemos que o botão "ver mais" fica do lado esquerdo da tela, enquanto deveria estar centralizado.

Para resolver isso, podemos aplicar um `justify-content` com o valor `center` no arquivo `flex.css`. O resultado ficará assim:

```
.eventos__lista {
  display: flex;
  flex-wrap: wrap;
  column-gap: 1.5rem;
  row-gap: 1rem;
  justify-content: center;
}
```

Em seguida, basta atualizar o projeto.

Já analisando a tela para tablets, vemos que é preciso inserir o mesmo botão no lado direito do layout. Para fazer essa modificação, usaremos o `justify-content` dentro do `@media` para a tela de tablet.

Para fazer isso, escreveremos:

```
.eventos__lista {  
    justify-content: end;  
}
```

Com o valor `end`, estamos informando que queremos que o elemento fique no fim da linha, ou seja, para o lado direito. Em seguida, atualizamos e redimensionamos a tela para 720 pixels. Com isso, podemos ver que o botão foi para o lado direito do layout.

Por fim, compararemos a seção de próximos eventos com o projeto pronto no Figma para verificar o que falta. A lista já está organizada com o flex-box, mas falta organizar os elementos dentro de cada li, ou seja, o nome, a data e o local de cada evento do lado esquerdo, além do botão "Ver", que deve estar no lado direito.

Poderíamos organizar os elementos do lado esquerdo aplicando-os dentro de uma `div` e inserindo um `flex-direction` com valor `column`. Isso faria com que ficassem alinhados na vertical, em coluna.

Já o botão "Ver" do lado direito ficaria fora dessa `div` e poderia ser organizado com alguma propriedade para `flex-item`. No entanto, seria difícil chegar a esse resultado de layout dessa forma. Nesse caso, o ideal é concluir com o `grid`.

Depois deste vídeo, deixarei um desafio para você praticar Flexbox: o de organizar os itens de cada li da parte "Coloque na sua agenda". O trabalho será parecido com a seção de próximos eventos que acabamos de editar.

Nesse desafio, deixarei todas as orientações para você fazer o exercício. Se tiver alguma dúvida, você pode postá-la o fórum.

## Flex

Hora de avançar no projeto testando seu conhecimento na prática!

Nos três tamanhos de tela em desenvolvimento, que são, mobile, tablet e desktop os elementos da lista agenda estão posicionados para o lado esquerdo do layout, um em cima do outro, dessa forma:



É preciso que você coloque em prática o que vimos nessa aula, deixando o layout com este resultado para **telas mobile**:

COLOQUE NA SUA AGENDA!



**AGOSTO**  
18 (QUINTA)  
20:30

Quinta 3D

[Avise-me](#)



**AGOSTO**  
16 (TERÇA)  
20:30

Festival de Cinema

[Avise-me](#)



**AGOSTO**  
18 (QUINTA)  
8 a 12:00

Dia de Balonismo

[Avise-me](#)



**AGOSTO**  
18 (QUINTA)  
20:30





Hard Rockers

[Avise-me](#)

[Ver mais](#)

Com este resultado para **telas de tablet**:







## COLOQUE NA SUA AGENDA!

 <p><b>AGOSTO</b> 18 (QUINTA) 20:30</p> <p>Quinta 3D</p> <p><a href="#">Avise-me</a></p>	 <p><b>AGOSTO</b> 18 (QUINTA) 20:30</p> <p>Quinta 3D</p> <p><a href="#">Avise-me</a></p>
 <p><b>AGOSTO</b> 18 (QUINTA) 8 a 12:00</p> <p>Dia de Balonismo</p> <p><a href="#">Avise-me</a></p>	 <p><b>AGOSTO</b> 18 (QUINTA) 20:30</p> <p>Hard Rockers</p> <p><a href="#">Avise-me</a></p>

[Ver mais](#)

Com este resultado para **telas de desktop**:

### COLOQUE NA SUA AGENDA!

 <p><b>AGOSTO</b> 08 (SEXTA) 20:30</p> <p>Quinta 3D</p> <p><a href="#">Avise-me</a></p>	 <p><b>AGOSTO</b> 08 (SEXTA) 20:30</p> <p>Festival de Cinema</p> <p><a href="#">Avise-me</a></p>	 <p><b>AGOSTO</b> 08 (SEXTA) 20:30</p> <p>Dia de Balonismo</p> <p><a href="#">Avise-me</a></p>
 <p><b>AGOSTO</b> 08 (SEXTA) 20:30</p> <p>Hard Rockers</p> <p><a href="#">Avise-me</a></p>	 <p><b>AGOSTO</b> 08 (SEXTA) 20:30</p> <p>Balé Municipal</p> <p><a href="#">Avise-me</a></p>	 <p><b>AGOSTO</b> 08 (SEXTA) 20:30</p> <p>Festival Multicores</p> <p><a href="#">Avise-me</a></p>

[Ver mais](#)

- Essa seção é composta por 7 itens, sendo 6 cards e 1 botão;
- O botão na tela para mobile fica no centro da lista e na tela de tablets e desktop deve ficar ao canto direito
- Utilize media query para estilizar o botão de tablets e desktop para telas a partir de 720 px de largura;
- Esse layout possui um espaçamento entre linhas de 1rem e um espaçamento entre colunas de 1.5rem;
- Todos os elementos da lista devem crescer proporcionalmente aproveitando a largura do container;
- O eixo principal desta lista deve ser horizontal;
- A solução é parecida com a construída na lista de eventos, realizada nesta aula.

#### • \\ código acima ocultado

```

• .agenda__lista {
•     display: flex;
•     flex-wrap: wrap;
•     row-gap: 1rem;
•     column-gap: 1.5rem;
•     justify-content: center;
• }
•
• .agenda__item {
•     flex-grow: 1;
• }
•
• @media (min-width: 720px) {
•     .agenda__lista {
•         justify-content: end;
•     }
• }

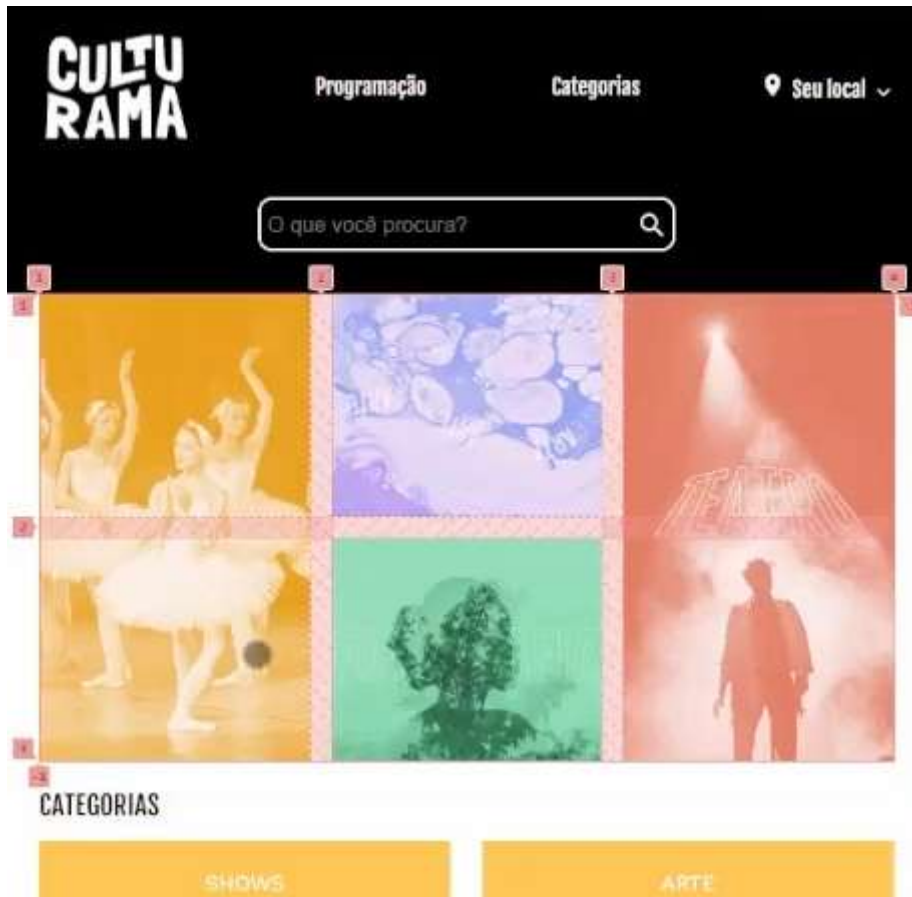
```

#### • \\ código abaixo ocultado

Repare que, no projeto finalizado no Figma, a primeira coluna do banner ocupa cerca de 50% da largura da página quando estamos em telas de tablets (720 pixels).

Os outros 50% restantes são distribuídos para a segunda e a terceira coluna. Entretanto, quando abrimos nosso projeto em desenvolvimento, não temos essa mesma proporção: cada coluna está ocupando cerca de 1/3 da largura do contêiner.





Para solucionar isso, acessaremos o arquivo `grid.css`. Na instrução `@media` que altera telas a partir de 720px, chamaremos a classe `.banner` e aplicaremos a ela a propriedade `grid-template-columns` recebendo a função `calc()`. Dentro dos parênteses, passaremos `50% - 0.75rem`, e após eles os valores `auto auto`.

```
@media(min-width: 720px) {
```

```
  .banner {
    grid-template-columns: calc(50% - 0.75rem) auto auto;
  }
```

//código omitido

Na linha abaixo, vamos modificar o gap, já que para esse tamanho de tela ele passará a ser 1.5rem.

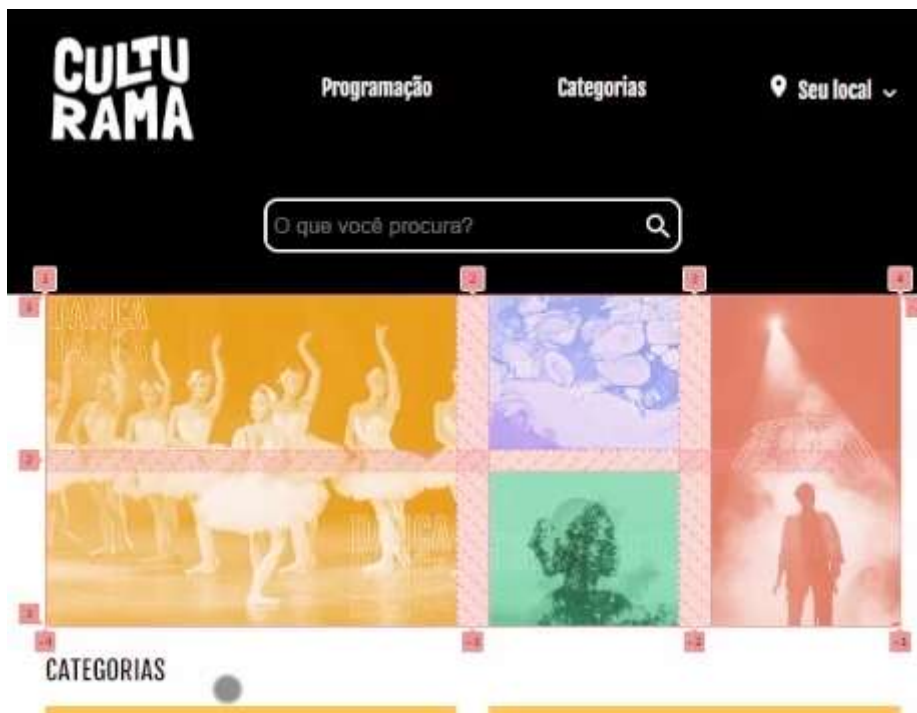
```
@media(min-width: 720px) {
```

```
  .banner {
    grid-template-columns: calc(50% - 0.75rem) auto auto;
    column-gap: 1.5rem;
```

}

//código omitido

Após salvar, nossas colunas terão a mesma proporção do projeto finalizado.



A propriedade `grid-template-columns` é responsável por determinar a largura de cada coluna.

Sendo assim, o primeiro valor, `calc(50% - 0.75rem)`, se refere à primeira coluna; e o segundo e o terceiro valor, `auto auto`, se referem, respectivamente, à segunda e à terceira coluna.

Quando utilizamos o valor `auto`, estamos instruindo o navegador a calcular automaticamente a largura para aquela coluna, levando em consideração os próprios elementos que estão ocupando esse espaço - ou seja, a segunda e a terceira coluna terão a mesma largura que as imagens que contêm.

Mas e o valor `calc(50% - 0.75rem)`? Para entender essa instrução, precisamos pensar que a largura total do contêiner - 640px - não é composta apenas pela largura das colunas, mas também pelos espaçamentos entre elas.

Isso é importante porque, se removermos o 0,75rem do nosso valor, mantendo apenas o 50%, o `grid-template-columns` força para que essa primeira coluna tenha 50% de largura, sem levar em consideração a existência dos espaçamentos aplicados no grid.

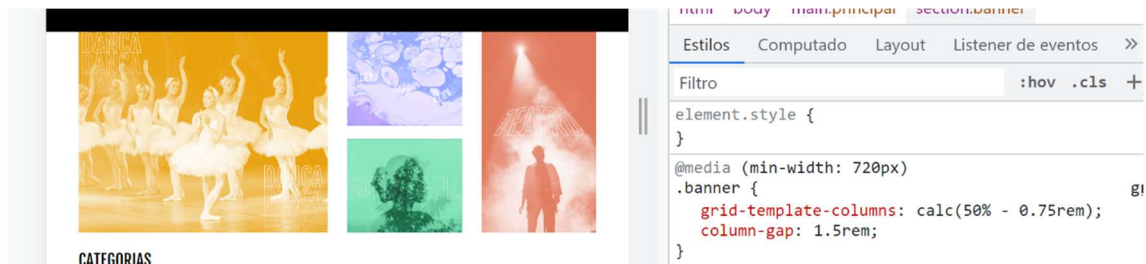
Para descontar esse espaçamento, utilizamos a função `calc()` do CSS para subtrair dos 50% o valor 0.75rem, referente ao tamanho do espaçamento (metade do 1.5rem que foi aplicado às duas colunas).

Agora o grid do banner está oficialmente finalizado, e contamos com você para enfrentar os desafios que virão pela frente!

Dentro do bloco do `@media (min-width: 720px)`, foi aplicado no banner, o seguinte código:

`grid-template-columns: calc(50% - 0.75rem) auto auto;`

Sendo que, se retirarmos os dois valores auto, veja só o que acontece:



### O resultado ficará o mesmo!

No vídeo anterior, foi aplicado os dois valores auto para que você pudesse entender melhor que no grid, cada valor se refere a uma determinada coluna, o primeiro valor é para a primeira, o segundo para a segunda e assim por diante.

Entretanto, **neste caso**, não há a necessidade de especificar que o valor é auto, isso porque quando não especificamos, o navegador já interpreta esse valor por padrão.

Sendo assim, o código do banner, para as telas a partir de 720px ficará:

```
.banner {
  grid-template-columns: calc(50% - 0.75rem);
  column-gap: 1.5rem;
}
```

### E por que neste caso?

Porque existem algumas situações que será necessário manter a escrita do valor auto.

Imagine que eu e você gostaríamos de aplicar o valor `calc(50% - 0.75rem)` na **segunda** ou **terceira** coluna ao invés da primeira. Como seria?

- Segunda
- ```
.banner {
  grid-template-columns: auto calc(50% - 0.75rem);
}
```
- Terceira
- ```
.banner {
  grid-template-columns: auto auto calc(50% - 0.75rem);
}
```

Agora, para que seja aplicada na coluna correta é indispensável a escrita do auto, visto que o navegador não tem como adivinhar sozinho onde o valor deve ser aplicado. Com isso, podemos chegar a conclusão que:

Se o valor auto for um **antecessor** do outro valor que está sendo aplicado, ele deverá estar declarado.

No Figma, desenhei em azul o grid de um elemento que em nosso projeto é um `<li>`. Vamos analisar os *grid items* alocados nele.

Na primeira linha temos uma imagem que ocupa a primeira e a segunda coluna. Na segunda linha, temos o nome do evento; na terceira, a data deste evento; e na quarta o local do evento na primeira coluna, além de um botão escrito "ver" na segunda coluna.



Esses elementos atuarão como *grid items*, mas também podem ser interpretados como áreas de um template no grid.

Analisando o projeto atual no navegador, vemos que os *grid items* não estão posicionados conforme o Figma. Vamos solucionar esse cenário de maneira um pouco diferente do que fizemos com o banner.

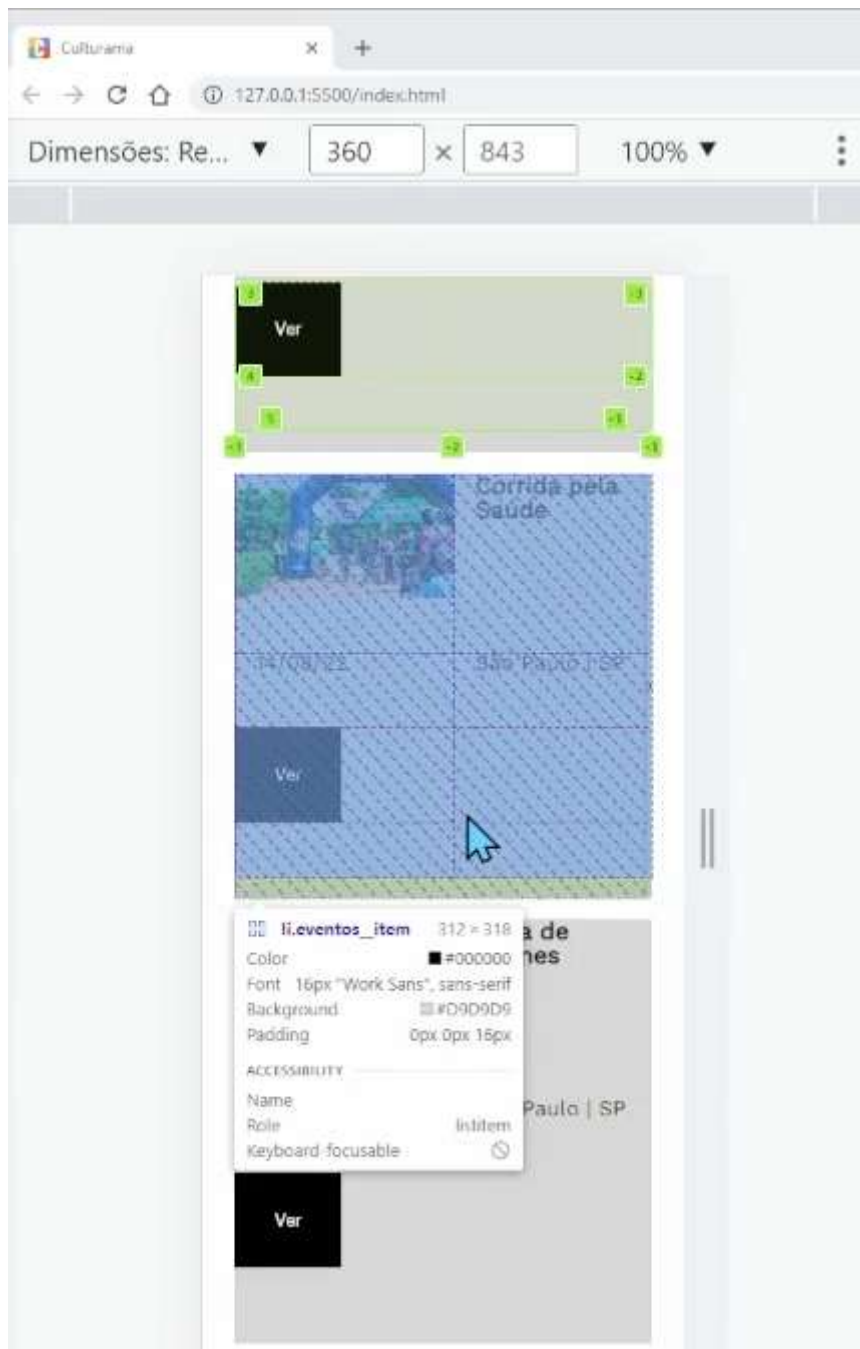
No arquivo `grid.css`, chamaremos a classe `.eventos__item`, referente à `<li>` desse elemento. Aplicaremos a ela a propriedade `display: grid`.

```
.eventos__item {  
    display: grid;  
}
```

Prosseguindo, incluiremos também a propriedade `grid-template-areas`. Passaremos a ela as instruções `imagem imagem`, `nome nome`, `data data` e `local botao`, todas entre aspas e uma em cada linha.

```
.eventos__item {  
  display: grid;  
  grid-template-areas:  
    "imagem imagem"  
    "nome nome"  
    "data data"  
    "local botao";  
}
```

Após salvar, nosso grid ficará bastante bagunçado na página, mas teremos criado duas colunas e quatro linhas.



A propriedade `grid-template-areas` cria e nomeia áreas um grid, determinando quantas linhas e colunas tais áreas ocuparão. A cada par de aspas duplas temos uma linha desse grid, e cada elemento representa uma coluna. A área nomeada como `imagem`, por exemplo, ocupará a primeira linha e a primeira e segunda coluna.

Apesar do grid já estar dividido da maneira desejada, os *grid items* ainda não estão na posição correta. Corrigiremos isso em breve!

Nós criamos as áreas do grid de nossa `<li>`, mas ficou faltando o principal: associar o elemento HTML a essa área.

Para isso, chamaremos a classe `.eventos__item` `img` e abriremos um bloco de chaves. Dentro dele, passaremos a propriedade `grid-area` com o valor `imagem`.

```
.eventos__item img {
    grid-area: imagem;
}
```

Em seguida, chamaremos a classe `.eventos__item h3`, dessa vez recebendo as propriedades `grid-area` com o valor `nome`.

```
.eventos__item img {
    grid-area: imagem;
}
```

```
.eventos__item h3 {
    grid-area: nome;
    align-self: center;
}
```

Prosseguiremos com `.eventos__item h4` recebendo a propriedade `grid-area` com o valor `data`; com `eventos__item h5` recebendo a propriedade `grid-area` com o valor `local`; e `.eventos__button` recebendo a propriedade `grid-area` com o valor `botao`.

```
.eventos__item img {
    grid-area: imagem;
}
```

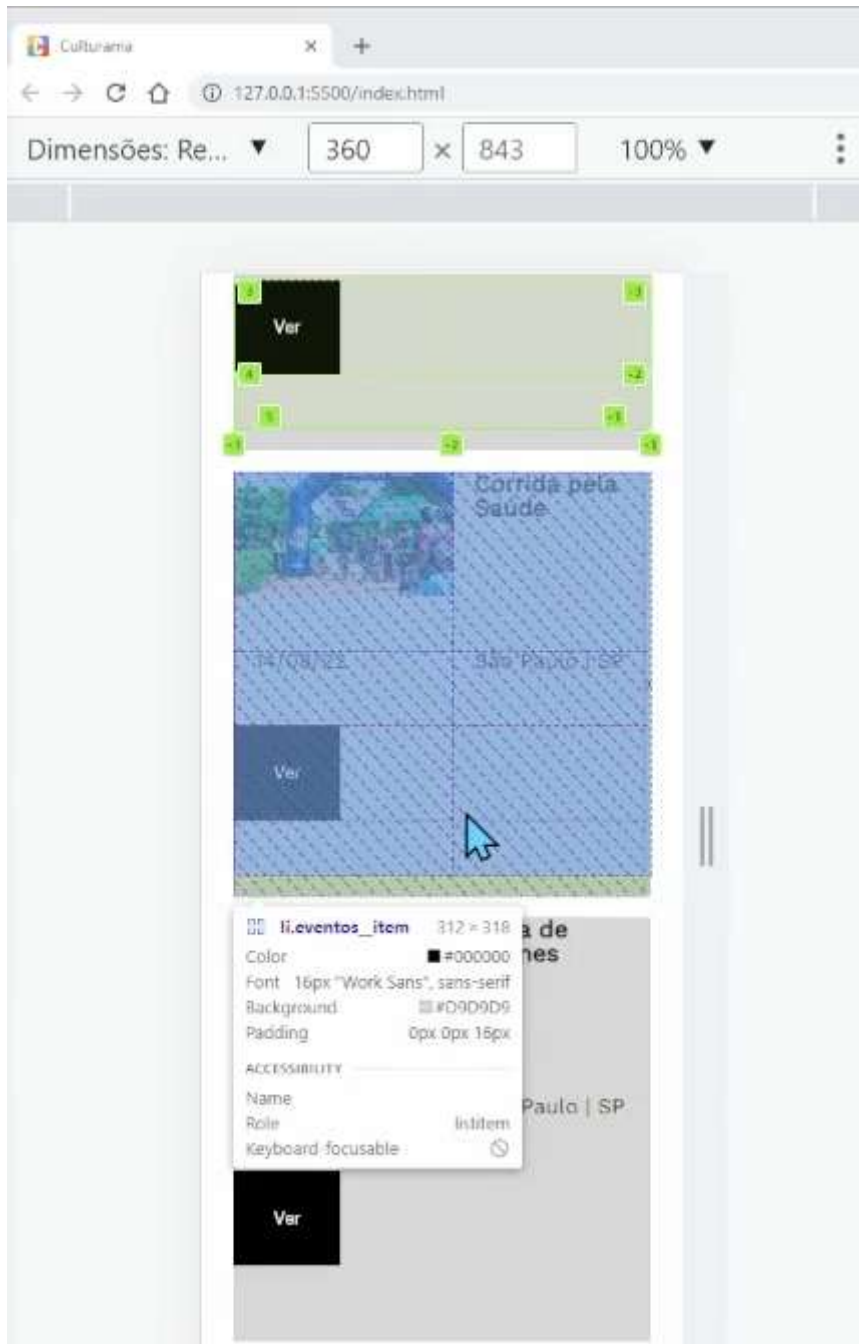
```
.eventos__item h3 {
    grid-area: nome;
}
```

```
.eventos__item h4 {
    grid-area: data;
}
```

```
.eventos__item h5 {
    grid-area: local;
}
```

```
.eventos__item button {
  grid-area: botao;
}
```

Feito isso, nossos *grid items* serão posicionados em seus devidos locais, assim como no projeto do Figma. Mas vamos prestar atenção: enquanto o *grid-template-areas* cria e nomeia as áreas, além de determinar quantas linhas e colunas elas terão, o *grid-area* carrega o nome dessa área e faz a associação entre o elemento e a área do grid.



No próximo vídeo vamos trabalhar com a dimensão entre as linhas e colunas.

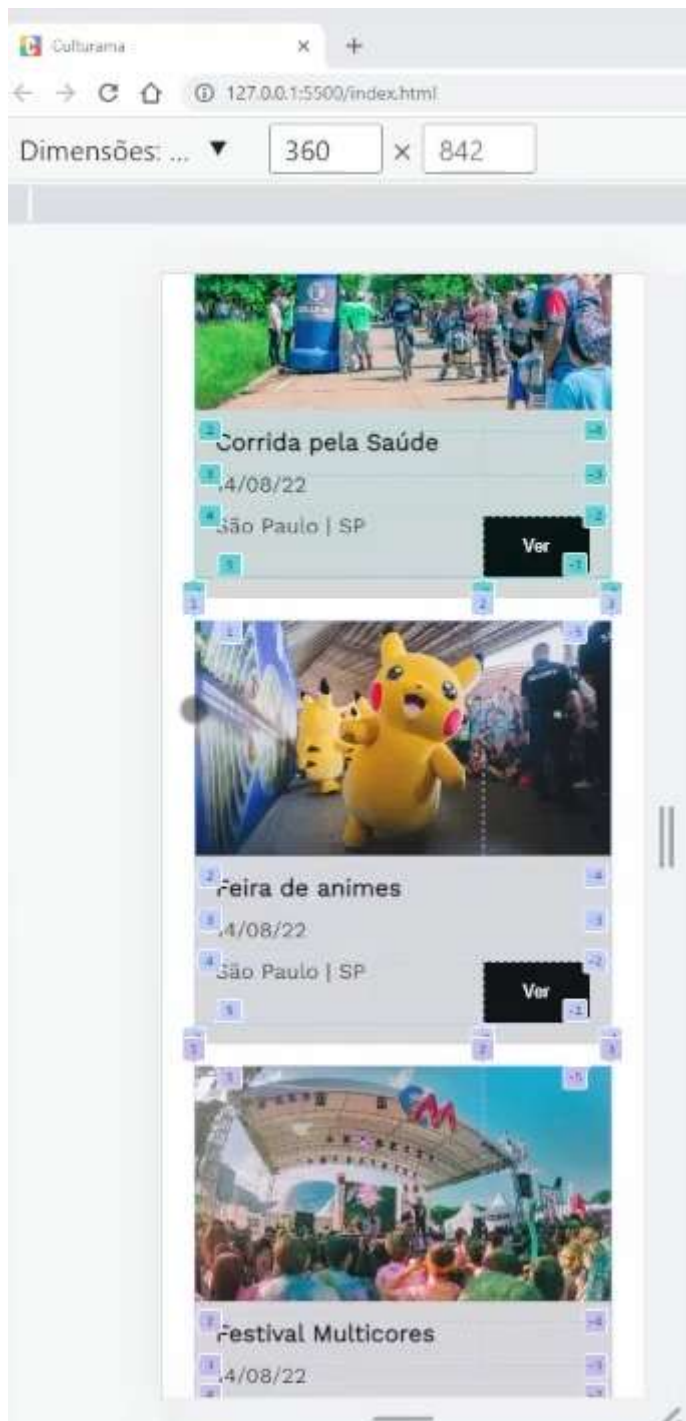


Com nosso projeto em desenvolvimento aberto, é possível reparar que a segunda coluna da <li> está com uma largura maior do que a primeira, e gostaríamos de inverter isso.

A ideia é que a segunda coluna tenha a mesma largura que a dos elementos que a ocupam, e que toda a largura restante do contêiner seja direcionada para a primeira coluna.

Para isso, acessaremos o arquivo grid.css. No bloco da classe .eventos\_\_item, incluiremos a propriedade grid-template-columns recebendo o valor 1fr auto.

```
.eventos__item {  
  display: grid;  
  grid-template-columns: 1fr auto;  
  grid-template-areas:  
    "imagem imagem"  
    "nome nome"  
    "data data"  
    "local botao";  
}
```



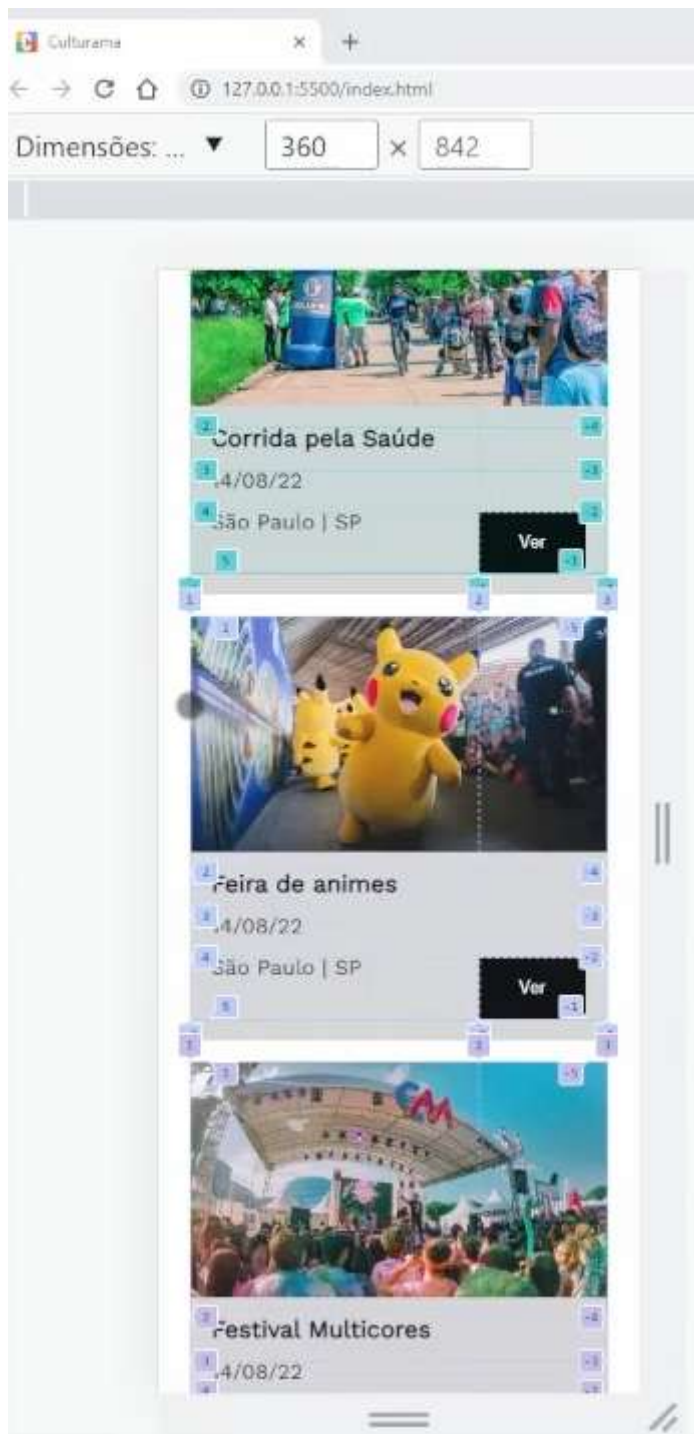
Feito isso, os elementos serão posicionados da maneira que desejamos. Já vimos anteriormente o que o valor `auto` significa, mas e o `1fr`?

O valor `fr` basicamente se refere a uma fração. quando utilizamos `1fr`, estamos indicando ao navegador que queremos uma fração do espaço restante. Dessa forma, uma parcela da largura do contêiner irá para o valor `auto`, e todo o restante para `1fr`.

Ainda precisamos ajustar as linhas. Analisando o projeto, percebemos que a primeira e a última linha estão com uma altura grande. Gostaríamos que, assim como nas colunas, a altura dessas linhas seja idêntica à altura dos elementos que as ocupam, enquanto a altura restante do contêiner seja igualmente distribuída para as demais linhas.

Para isso, incluiremos no bloco da classe `.eventos__item` a propriedade `grid-template-rows` recebendo o valor `auto repeat(2,1fr) auto`.

```
.eventos__item {  
  display: grid;  
  grid-template-columns: 1fr auto;  
  grid-template-rows: auto repeat(2, 1fr) auto;  
  grid-template-areas:  
    "imagem imagem"  
    "nome nome"  
    "data data"  
    "local botao";  
}
```



Feito isso, os elementos na página serão ajustados. Assim como a propriedade `grid-template-columns` determina a largura de cada coluna, a `grid-template-rows` determina a altura de cada linha.

Sua utilização segue a mesma lógica: o primeiro valor se refere à primeira linha, o segundo valor à segunda linha e assim por diante. Sendo assim, determinamos o valor `auto` para a primeira e para a última linha.

Já a segunda e a terceira linha receberão o valor `1fr`. Note que, neste caso, utilizamos a função `repeat()` do CSS. Tal função recebe como parâmetro dois valores: o primeiro é referente a quantas vezes queremos repetir, e o segundo é o valor que será aplicado repetidas vezes.

O interessante é que a função `repeat()` compila esses dois valores em uma única informação, e podemos usá-la sempre que desejamos ter uma repetição sequencial de valores, o que é exatamente o cenário em que estamos trabalhando.

Após esse processo, conseguimos cumprir um dos maiores objetivos do grid, que é trabalhar simultaneamente com linhas e colunas - ou seja, o eixo X e o eixo Y.

Para praticar ainda mais a manipulação da altura das linhas e largura das colunas, é importante que você realize a próxima atividade, um desafio que será pré-requisito para os conteúdos a seguir. Bons estudos!

### Grid da li

Você não achou que deixaríamos de fora o grid da `<li>` da agenda, não é mesmo?

Chegou o momento de você colocar em prática tudo que aprendeu nesta aula e praticar em um desafio.

No momento, a `<li>` se encontra dessa forma:



E esse é o resultado que você irá chegar!



Algumas informações para te auxiliar.

- Esse layout é composto de 2 colunas e 2 linhas;
- A largura total do Grid Container foi fracionada igualmente para as duas colunas;
- A primeira linha tem a altura idêntica ao do elemento que a ocupa e toda altura restante do container foi direcionada para a segunda linha;
- Caso você queira compactar a sintaxe, utilize como aliada a função `repeat()`;
- O grid item que é uma imagem, deverá ocupar as colunas 1 e 2, do traço 1 até o traço 3
- 

### • Grid Container

- O primeiro passo para solucionar esse desafio é ir ao arquivo `grid.css` e chamar o elemento que atuará como Grid Container, para isso, é necessário trazer o contexto de grid com o `display`.

```
.agenda__item {
  display: grid;
}
```

### • Grid Item

- Em seguida, é necessário posicionar o grid item em seu devido lugar, a informação que temos é que a imagem deverá ocupar as colunas 1 e 2, do traço 1 até o traço 3.

```
.agenda__item img {
  grid-column: 1/ 3;
}
```

### • Dimensionando a altura das linhas e largura das colunas

- Agora, já podemos trabalhar nas linhas e colunas, primeiro precisamos criar duas colunas que tenham a mesma proporção.

```

• .agenda__item {
•   display: grid;
•   grid-template-columns: repeat(2, 1fr);
• }

```

- E por último, faremos com que a primeira linha tenha exatamente a mesma altura da imagem e o restante da altura seja direcionada para a segunda linha.

```

• .agenda__item {
•   display: grid;
•   grid-template-columns: repeat(2, 1fr);
•   grid-template-rows: auto 1fr;
• }

```

Você venceu! Parabéns! O grid da li está finalizado.

Vamos voltar à <li> de eventos. Ficou faltando ajustar um último detalhe, que pode ser resolvido com uma funcionalidade *plus* do *grid*. Repare que a <li> tem um título, "Festival das luzes". Esse título é um h3 e se encontra alinhado verticalmente, no topo da área. Queremos que ele fique alinhado verticalmente no centro.

O mesmo ocorre com o título "São Paulo | SP", que é um h5. Ele se encontra alinhado verticalmente no topo e gostaríamos que ele ficasse alinhado verticalmente no final da área. Como solucionamos isso?

No grid.css, dentro do bloco de .eventos\_item h3, abaixo de grid-area, escreveremos align-self: center;.

```

.eventos_item h3 {
  grid-area: nome;
  align-self: center;
}

```

No bloco de .eventos\_item h5, escreveremos align-self: end.

```

.eventos_item h5 {
  grid-area: local;
  align-self: end;
}

```

Basta salvar e teremos o resultado desejado. Tudo isso graças à propriedade **align-self**, que é responsável por alinhar individualmente um *grid item* ou um *flex item*. Assim como o *gap*, a propriedade *align* também é compartilhada entre o *flex* e o *grid*. É isso que estudaremos no *flex* do Flutter com o .

Agora vamos finalizar a última sessão do projeto **Culturama**. Começaremos abrindo a opção "Inspeccionar" (botão direito do mouse) e alterando, em "Dimensões: Responsivo", a largura para 360 pixels, já que estamos configurando primeiro a sessão para celulares.

A última sessão é a de rodapé. Vamos comparar com o nosso objetivo, assim saberemos o que precisa ser feito. Ao realizarmos a comparação, perceberemos que no nosso projeto, os elementos estão todos se sobrepondo. Precisamos deixá-los alinhados em níveis e que eles passem sempre para a próxima linha quando não couberem no mesmo nível.

Vamos procurar o código do rodapé no nosso projeto. Ele começa na linha 212, `<footer class "rodape">`, e vai até a 268, `</footer>`. Também abriremos o arquivo "flex.css" e vamos codar. Na linha 60, vamos inserir um comentário, `/* Rodapé */` para que o código fique organizado. Mais acima, temos os códigos da área de agenda que foram passados como desafio. Caso você ainda não tenha feito, pode voltar nas últimas atividades.

O próximo passo é conferir o nosso código HTML. Os elementos do rodapé estão dentro de `<ul>`. Existem várias `<ul>` e, ao final, uma `div`. Nessa `div`, temos um `<h4></h4>` e seu `background` é a logo do site, "Culturama" e também a frase `<p>Desenvolvido por SENAI.</p>`. Vamos organizar os elementos da `<ul class="rodape_lista">`.

Nós queremos que o Flexbox entenda que cada grupo de `<ul>`, junto da `div`, é um item. Para organizar todos esses itens, localizaremos a origem deles, o "pai", no HTML. Essa origem é a `tag <footer class="rodape">`, com a classe "rodapé". Nós levaremos essa classe para o arquivo "flex.css".

No arquivo "flex.css", vamos escrever essa classe na linha 62:

```
.rodape{

}
```

Agora vamos deixar o projeto do VSCode à esquerda da tela e o projeto do navegador à direita, assim poderemos compará-los. Vamos aplicar o `display: flex`; e conferir.

```
.rodape {

    display: flex;

}
```

Os elementos ficaram dispostos lado a lado. Ainda está faltando o efeito de quebra de linha. Para isso, aplicaremos o `flex-wrap`: com o valor `wrap`.

```
.rodape {

    display: flex;

    flex-wrap: wrap;

}
```



O nosso rodapé está ficando bem legal! Os elementos não estão mais amontoados em uma mesma linha. Sempre que isso vai acontecer, o elemento passa para a próxima linha.

Comparando com o rodapé do Figma, no projeto pronto, notamos que está faltando aplicar espaçamento entre os itens. Nós até poderíamos aplicar um `column-gap` para criarmos um espaçamento entre colunas, mas utilizaremos o `justify-content`: com o valor `space-between`.

```
.rodape {
  display: flex;

  flex-wrap: wrap;

  justify-content: space-between;
}
```

Vamos atualizar a página. Agora o espaçamento está aplicado corretamente entre os elementos. Vamos aumentar o tamanho das dimensões e testar, com o "Inspeccionar", como o texto é distribuído quando a largura do container aumenta.

O resultado com o `column-gap` não seria bom, porque ele é uma medida estática. Com o `justify-content`, temos um efeito muito melhor. Vamos localizar o rodapé pelo "Inspeccionar" para visualizarmos esse efeito.

Passando o mouse em cima do `justify-content` no "Inspeccionar", perceberemos que todo o espaçamento entre os itens, representado em roxo na imagem, é um efeito aplicado pelo `justify-content`.

Se "desselecionamos" essa opção, os elementos são jogados para o início da linha. Quando voltamos com o `justify-content`, notamos que os elementos são organizados automaticamente, aproveitando todo o espaço disponível dentro do container do rodapé. Para o nosso próximo passo, vamos alterar outra vez a largura do *layout* pelo "Inspeccionar" para 360 pixels.

Nós temos uma logo, "Culturama", e abaixo, a frase "Desenvolvido por SENAI". Precisamos descer esses elementos para o final da linha onde estão inseridos e na vertical, de cima para baixo. Se é na vertical, significa que está fora do eixo principal, então, usaremos uma propriedade que modifica a posição dos elementos fora do eixo principal.

Vamos procurar esses elementos no HTML. Eles estão na linha 264 até a 267, dentro da `<div class="rodape_logo">`. Usaremos a classe dessa div para organizar esses dois elementos de uma só vez.

Na linha 68, traremos a classe do arquivo `flex.css`. Escreveremos `.rodape__logo {` e adicionaremos uma propriedade de `flex-item` que altere a posição dos elementos fora do eixo principal. É a mesma propriedade que utilizamos algumas vezes no *grid*, a `align-self`. Ela também pode ser utilizada para Flexbox. Além da `align-self`, existem outras, como a `justify-content`, a `column-gap` e a `row-gap`.

Como desejamos que esses elementos fiquem no final da linha, vamos inserir o valor `end` e atualizar.

```
.rodape__logo {
  align-self: end;
}
```

Finalizamos o projeto para tela de celulares. Vamos verificar para tela de tablets, alterando a dimensão para 720 pixels. Está perfeito. Agora, para telas de desktop. Perfeito também! Terminamos o projeto Culturama.

Assim como a propriedade `Gap`, que é utilizada dentro dos contextos de `Grid` e `Flexbox`, a propriedade `Align` também é compartilhada entre eles, porém existem algumas diferenças em seu uso!

### Alinhamento no Grid

Para gerar um alinhamento **vertical** dentro do `Grid`, utiliza-se a propriedade `align`. E para criar um alinhamento *horizontal*, usa-se a propriedade `justify`. Ambas são acompanhadas de outras propriedades que vão dizer se o alinhamento será de um único item, todos os itens ou do container.

### Alinhamento no Flex

No `flex`, não utiliza-se o `justify` para alinhar horizontalmente como no `grid`, o alinhamento no `flex` irá ser determinado pela propriedade `flex-direction`. Se for o padrão, ou seja, `flex-direction: row`, o `align` irá alinhar na direção **vertical** e caso seja `flex-direction: column` que está sendo aplicado, `align` irá alinhar na direção **horizontal**.

### Grid da seção de categorias

Chegou o momento de você colocar em prática tudo que aprendeu neste treinamento sobre `Grid CSS` em um último desafio, o `grid` da seção de categorias para as telas a partir de 1440px!

Esse é o resultado desejado que você irá atingir!



A seguir irei colocar algumas instruções que irão te ajudar a solucionar esse desafio, vamos lá!

1. Não esqueça de criar um bloco com `@media (min-width: 1440px) {}` ao final do arquivo `grid.css` para alocar todo o código que você irá construir.

2. Esse layout possui um espaçamento entre linhas de 1rem e um espaçamento entre colunas de 1.5rem.
3. O layout é composto de 2 colunas e 3 linhas.
4. Em relação a largura, terá a mesma proporção e lógica que o banner teve para as telas a partir de 720px, onde foi utilizado a função calc.
5. Já as linhas, a primeira e a última possuem a mesma altura dos elementos que ocupam e a linha do meio possui a altura restante que sobrou do container.
6. Para posicionar cada grid item, você pode utilizar as grid lines (grid-row / grid-column), mas recomendo criar áreas no template e nomeá-las (grid-area).

## 7. PASSO 1: Criando o @media

8. Dentro de grid.css, ao final de todo o código, crie o bloco que irá conter apenas propriedades que serão aplicadas a partir da tela de 1440px.

```
9. @media (min-width: 1440px) {
10. }
11.
```

## 12. PASSO 2: Trazendo o grid para o container

13. Dentro do bloco, nós não conseguimos fazer nada antes de trazer o contexto do grid para o container que estamos trabalhando. O código ficará assim:

```
14. @media (min-width: 1440px) {
15.     .categorias {
16.         display: grid;
17.     }
18.
```

## 19. PASSO 3: Criando as áreas do template

20. Bora analisar novamente a imagem do modelo!



- 21.
22. A primeira linha do grid é composta completamente pelo **título** da seção. A segunda linha é compartilhada com a **lista** de categorias, que ocupa a primeira coluna e a seção de **destaques**, que ocupa a segunda coluna. Já na terceira e última linha, há novamente a lista de categorias e a **imagem** que ocupa a segunda coluna.

23. Com essas informações conseguimos criar um template com áreas nomeadas, o código ficará assim:

```
24. @media (min-width: 1440px) {
25.     .categorias {
26.         display: grid;
27.         grid-template-areas:
28.             "titulo titulo"
29.             "lista destaque"
30.             "lista imagem";
31.     }
32.
```

### 33. PASSO 4: Posicionando os grid items

34. Com as áreas criadas, já é possível associar o elemento HTML àquela área do grid.

35. Abaixo do bloco de `.categorias`, crie novos blocos para cada elemento:

```
36. .categorias__titulo {
37.     grid-area: titulo;
38. }
39.
40. .categorias__lista {
41.     grid-area: lista;
42. }
43.
44. .categorias__destaques {
45.     grid-area: destaque;
46. }
47.
48. .categorias__imagem {
49.     grid-area: imagem;
50. }
51. }
52.
```

### 53. PASSO 5: Adicionando os espaçamentos

54. Segundo as instruções, esse layout possui um espaçamento entre linhas de 1.5rem e um espaçamento entre colunas de 1.5rem

```
55. @media (min-width: 1440px) {
56.     .categorias {
57.         display: grid;
58.         grid-template-areas:
59.             "titulo titulo"
60.             "lista destaque"
61.             "lista imagem";
62.         row-gap: 1rem;
63.         column-gap: 1.5rem;
64.     }
65.
```

### PASSO 5: Calculando a largura das colunas e a altura das linhas

Em relação a largura, essa seção irá funcionar da mesma maneira que aconteceu anteriormente com o banner, ou seja, a

largura de colunas é 50% mas é necessário levar em consideração e descontar o gap proporcional, ou seja, se aplicarmos um column-gap de 1rem, o proporcional será metade (0.75rem).

```

66. @media (min-width: 1440px) {
67.   .categorias {
68.     display: grid;
69.     grid-template-areas:
70.       "titulo titulo"
71.       "lista destaque"
72.       "lista imagem";
73.     row-gap: 1rem;
74.     column-gap: 1.5rem;
75.     grid-template-columns: calc(50% - 0.75rem);
76.   }
77.

```

Por último, as alturas das linhas já estão quase idênticas ao do modelo, pois todas estão como padrão ocupando uma altura `auto` e isso faz com que sobre um espaço de altura bem pequeno embaixo do container, veja onde a seta azul está apontando:



78.  
79. E o desejado é direcionar esse pequeno espaço de sobra para a segunda coluna, utilizando 1fr.

```

80. @media (min-width: 1440px) {
81.   .categorias {
82.     display: grid;
83.     grid-template-areas:
84.       "titulo titulo"
85.       "lista destaque"
86.       "lista imagem";
87.     row-gap: 1rem;
88.     column-gap: 1.5rem;
89.     grid-template-columns: calc(50% - 0.75rem);
90.     grid-template-rows: auto 1fr auto;
91.   }
92.

```

Aeeeeee, olha só onde você chegou, bora celebrar!

Neste treinamento, você venceu todos os desafios, enfrentou as batalhas e mergulhou nos estudos de CSS Grid e Flexbox para construir a página do Culturama.

Agora é sua vez de colocar em prática todo o conhecimento adquirido. Fique livre para personalizar o seu projeto da maneira que desejar e compartilhar no github, linkedin e onde mais desejar! Não deixe de nos marcar, vamos adorar ver como ficou!

**Parabéns!!!**