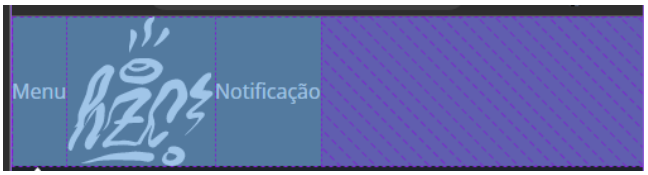
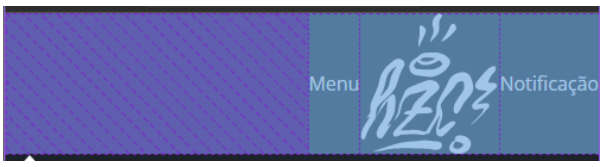
CSS – Dispondo elementos com Flexbox e Grid

Link do figma, onde está todo o nosso conteúdo e layout do site: https://www.figma.com/file/ibWktwVpnog76rMYOdVhks/Dispondo-elementos-com-flexbox-e-grid?node-id=54%3A2358.

1. **Aula 1 – Flexbox:**
    1.1. O projeto e as Ferramentas:
        1.1.1. Todas as informações que precisamos saber sobre cores, fontes e afins, além de estar no figma colo descrito acima, encontram-se no arquivo readme da pasta.
    1.2. O Flex Container:
        1.2.1. Convertemos o nosso cabeçalho para flex container com o display: flex;.
    1.3. Guia completo de flexbox | CSS tricks: https://css-tricks.com/snippets/css/a-guide-to-flexbox/.
    1.4. Justify-content e align-items:
        1.4.1. Ambos são propriedades que alteram o posicionamento dos elementos em um flex container:
        1.4.2. Justify-content:
            1.4.2.1. Flex-start: Todos os elementos ficam o mais próximo da esquerda possível:



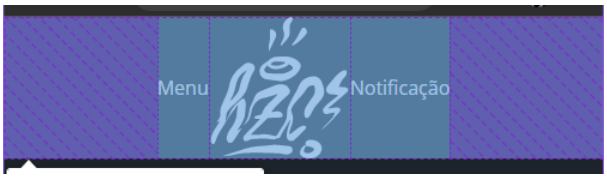
            1.4.2.2. Flex-end: O contrário do acima:



            1.4.2.3. Space-between: Todos os elementos são distribuídos igualmente dependendo do espaço no container, onde o primeiro elemento sempre estará mais a esquerda e o último sempre mais a direita possível.

1.4.2.4. Sapce-around: Igual ao acima, mas o espaço sobrando no container se divide também antes do primeiro elemento e depois do último, ou seja, os elementos são separados e espaçados no meio do container. Não distribui o espaçamento igualmente entre os elementos, portanto, entre o menu e a logo tem um espaçamento diferente de entre o menu e a borda da página:

1.4.2.5. Center: Joga tudo pro centro



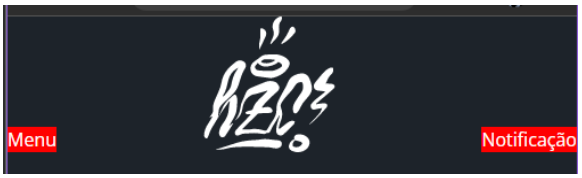
1.4.2.6. Space-evenly: Parecido com o around, mas distribuí o espaçamento igualmente entre os elementos:

1.4.3. Align-items: Responsável pelo alinhamento do conteúdo da flex container:

1.4.3.1. Center: Deixa todos alinhados no centro do container:

1.4.3.2. Flex-start: Todos para o topo to container:

1.4.3.3. Flex-end: O contrário do acima:



1.4.3.4. Stretch: Faz os elementos ocuparem a altura toda do container:

1.5. O que aprendemos:

1.5.1. O que é um flex-container:

1.5.1.1. Flex container é o elemento que recebe grande parte das propriedades de posicionamento para suas tags filhas.

1.5.2. As propriedades de posicionamento justify-content e align-items:

1.5.2.1. justify-content distribui o espaço restante do flex container entre suas tags filhas e align-items alinha verticalmente as tags filhas, ou seja, são propriedades de posicionamento horizontal e vertical respectivamente.

2. Aula 2 – Mais Funcionalidades do Flexbox:

2.1. Terminando o Cabeçalho:

2.1.1. Adicionamos ícones no lugar dos escritos menu e notificação utilizando para isso a font que está na pasta font do assets.

2.1.2. Para utilizar esses ícones precisamos colocar uma tag sendo essa uma tag de pseudo-elemento, para que ela seja substituída pelo ícone que desejamos colocar:

2.1.3. Para importar a font fazemos isso no próprio CSS com a anotação:

```
@font-face {
```

```
    font-family: 'icones';
    src: url(../font/icones.ttf);
}
```

2.1.4. E para substituir a tag <i> pelo ícone desejado, basta colocar um i::before{contente: ”\número do ícone encontrado no site”}:

```
.cabecalho__notificacao i::before {
  content: "\e906";
  font-size: 24px;
}
```

2.1.5. Dessa forma ao invés do escrito ‘notificação’ agora temos um sininho. Como são fontes, podemos alterar o tamanho delas com o font-size normalmente.

2.1.6. Como esses ícones foram colocados em botões no HTML, fica assim no resultado final:

```
<button class="cabecalho__menu" aria-label="Menu"><i></i></button>
```

2.1.7. O aria-label=”Menu” serve para indicar para os leitores de tela que isso é um botão de menu, já que não tem mais o escrito dentro dele, somente a tag <i>. **Essa é uma correção de acessibilidade que precisamos fazer sempre que formos utilizar essa técnica!!!!**

2.2. Para saber mais sobre esse tipo de ícone na web: https://www.alura.com.br/artigos/como-utilizar-icones-em-paginas-web.

2.3. O flex-wrap e flex-direction:

2.3.1. Durante a utilização do Emmet (facilitador de escrita), podemos colocar [atributo=”valor”] para definir o valor enquanto estamos escrevendo para não ter que fazer isso manualmente depois:

```
nav.menu-lateral>img.menu-lateral__logo+a[href=#].menu-lateral__link*6
```

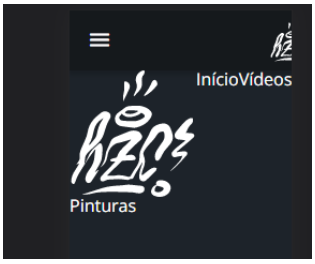
Resultado:

```
<nav class="menu-lateral">
    <img src="assets/img/logo.svg" alt="" class="menu-lateral__logo">
    <a href="#" class="menu-lateral__link">Início</a>
    <a href="#" class="menu-lateral__link">Vídeos</a>
    <a href="#" class="menu-lateral__link">Picos</a>
```

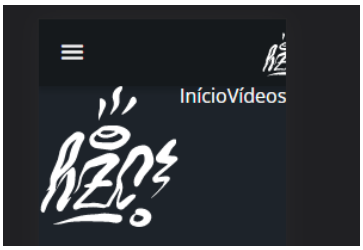
```
    <a href="#" class="menu-lateral__link">Integrantes</a>
    <a href="#" class="menu-lateral__link">Camisas</a>
    <a href="#" class="menu-lateral__link">Pinturas</a>
</nav>
```

2.3.2. Flex-wrap: quebra de linha dos elementos:

2.3.2.1. Wrap: Efetua a quebra de linha:



2.3.2.2. Nowrap: Impede a quebra de linha. Geralmente é o padrão:



2.3.2.3. Wrap-reverso: :

2.3.3. Felx-direction: Determina a direção que nossos elementos irão seguir:

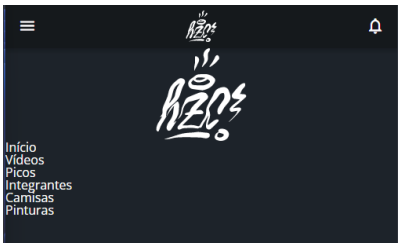
2.3.3.1. Row: Linhas. É o que vem por padrão quando colocamos o flex como display. Ele irá alocar todos os elementos um ao lado do outro.

2.3.3.2. Column: Colunas. Aloca todos os elementos um abaixo do outro:

2.3.3.3. Quando invertemos a direção, os valores de justify-content e align-items também são invertidos!!!

2.3.4. Quando convertemos uma sessão para flexbox, todos os elementos dentro dela também se tornam caixas, portanto, cada item possuí propriedades similares a box toda, como align-items, mas, como se trata somente de 1 elemento, utilizamos o align-self para ele:

```
.menu-lateral__logo {
  align-self: center;
}
```



2.3.4.1. Podendo assim ajustar somente a logo no centro, por exemplo. Lembrando que só ajustamos na horizontal utilizando o align e não o justify porque invertemos a direção do flexbox com o flex-direction, logo, todos os valores de direção são invertidos junto. Para ter esse efeito com o flex-direction em row, precisaríamos ter usado o justify-self e não o align-self como foi feito.

2.4. Para saber mais: Jogo do sapo para praticar flexbox: https://flexboxfroggy.com/.

2.5. Sem Medo de Usar Flex:

2.5.1. Os pseudo-elementos são aqueles que criamos utilizando o ::before ou ::after em uma classe de CSS.

2.5.2. Ao criar um pseudo-elemento em uma classe, podemos automaticamente transformar essa classe em um flexbox e manejar todos os pseudo elementos dentro dela como em um container normalmente:

Antes do pseudo elemento:

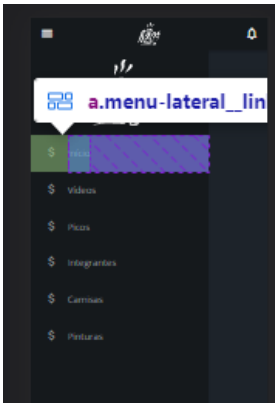
```
.menu-lateral__link {
  color: #95999C;
```

```
  height: 64px;
  padding-left: 64px;
}
```

Depois do pseudo:

```
.menu-lateral__link {
  align-items: center;
  color: #95999C;
  display: flex;
  height: 64px;
  padding-left: 64px;
}

.menu-lateral__link::before{
  content: "\e905";
  font-size: 24px;
  height: 24px;
  left: 24px;
  position: absolute;
  width: 24px;
}
```



2.5.3. Ao transformar a tag <a> em um flex, controlamos tanto o pseudo elemento ::before quanto o escrito dela, tornando ambos elementos:

```
▼<a href="#" class="menu-lateral__link">flex == $0
    ::before
    "Início"
  </a>
```

2.5.4. Criamos um pseudo-elemento para o menu-lateral__link, mas se quisermos que cada link tenha esse pseudo link alterado, precisamos criar um pseudo para cada e, como cada link terá um ícone diferente, precisamos criar uma classe separada para cada um:

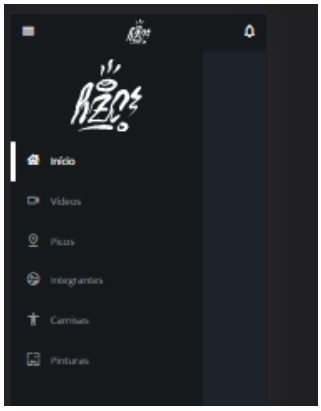
```
<a href="#" class="menu-lateral__link menu-lateral__link--inicio menu-
lateral__link--ativo">Início</a>


```CSS:
.menu-lateral__link--inicio::before {
  content: "\e902";
}
```

2.5.4.1. Coloquei como exemplo do link de início, mas isso se descorreu para todos os outros links também.

2.5.5. A classe –ativo é correspondente a alteração que cada link sofrerá quando mudarmos a página, ou seja, se estivermos na página de início, isso ocorrerá com o link:

```
.menu-lateral__link--ativo {
  color: #FFFFFF;
  padding-left: 56px;
  border-left: 8px solid #FFFFFF;
}
```


```

2.5.5.1. O mesmo se segue caso mudemos de página para vídeos, integrantes, picos e afins.

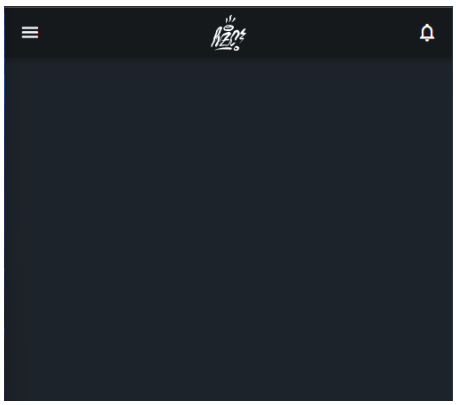
2.6. Lançando o Desafio:

2.6.1. Fazer com que o menu lateral apareça ou não... Basicamente colocamos na classe padrão dele a posição absoluta e o left como -100vw para ter certeza que não será exibido, em seguida criamos uma outra classe chamada de menu—ativo e voltamos o valor left para 0.

```css
.menu-lateral {
  background-color: #15191C;
  display: flex;
  flex-direction: column;
  height: 100vh;
  width: 75vw;

  position: absolute;
  left: -100vw;
  transition: all, .40s;
}

.menu-lateral--ativo {
  left: 0;
  transition: all, .40s;
}
```

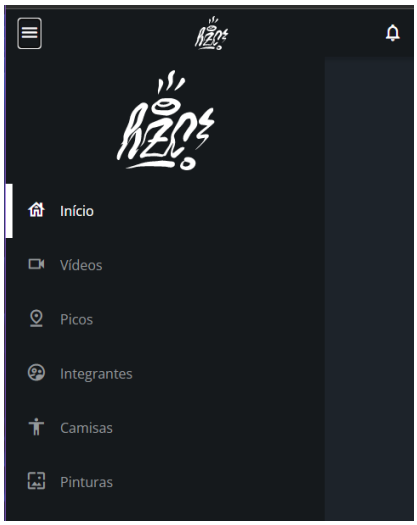


2.6.1.1. Colocamos o transition para que ele tenha um efeito sempre que aparecer e desaparecer o menu lateral.

2.6.2. No JS colocamos um addEventListener() no botão do menu e dizemos que ao ser clicado, ele irá toggle() a classe menu—ativo, ou seja, sempre que o botão for apertado ele irá adicionar ou remover a classe, dependendo do estado anterior:

```
const botaoMenu = document.querySelector('.cabecalho__menu')
const menu = document.querySelector('.menu-lateral')

botaoMenu.addEventListener('click', () => {
  menu.classList.toggle('menu-lateral--ativo')
})
```



2.7. O que aprendemos:

2.7.1. Como criar quebra de linha de um flex-container com a propriedade flex-wrap:

2.7.1.1. flex-wrap é a propriedade que usamos quando não existe mais espaço para comportar todos os elementos horizontalmente/verticalmente e é necessário uma “quebra de linha” para manter a proporção dos elementos.

2.7.2. Alterar a orientação do flex container com a propriedade flex-direction:

2.7.2.1. Naturalmente a orientação do flex container é na horizontal e para trocar o seu eixo, basta usar a propriedade flex-direction.

2.7.3. Propriedades de posicionamento de um flex-item com justify-self e align-self:

2.7.3.1. As propriedades de posicionamento justify-content e align-items movimentam todos os flex items, se precisamos de um posicionamento individual, usamos as propriedades -self nos flex items específicos.

2.7.4. Uso de flex para remanejar pseudo-elementos:

2.7.4.1. Existem diversas situações que a propriedade de flex pode ser utilizada. É inclusive um incentivo usar flex no lugar de trocar o display para inline/inline-block.

3. Aula 3 – O Que o Flex Não Consegue Fazer:

3.1. Limitações do Flexbox:

3.1.1. Para adicionar mais de uma classe em uma tag enquanto escreve no Emmet é só ir separando elas por '.'.

3.1.2. Se precisarmos que apenas 2 elementos em uma divisão fiquem um ao lado do outro, não conseguimos usar o flex, uma vez que ele age no parente mais próximo dele, que nesse caso seria uma divisão com 5 itens e alterando até o que não queremos.

3.1.3. Uma solução seria criar uma div para englobar somente esses 2 elementos, mas aí começa a se tornar uma gambiarra e não algo adequado de se fazer.

3.2. A solução do Grid:

3.2.1. Funciona similar ao flex, portanto podemos reutilizar todos os fundamentos que vimos até agora.

3.2.2. Funciona como um grid container, assim como o flex, portanto dizemos que uma div é um grid através do display.

3.2.3. Por padrão o display grid deixa um abaixo do outro, inverso ao flex.

3.2.4. As mesmas propriedades do flex funcionam para o grid.

3.2.5. Como o grid joga todos para colunas, o justify-content space-aroud e between acabam servindo como um flex-start, já que não existem outros elementos na mesma linha que eles para que o espaço seja remanejado.

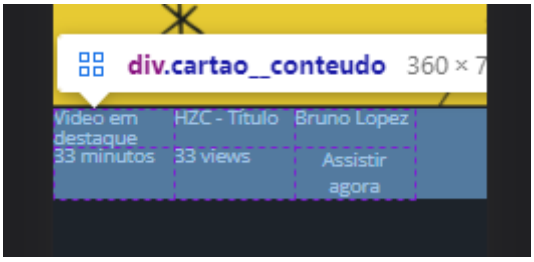
3.2.5.1. No final acabamos por usar somente 3 valores: felx-start/end ou center.

3.2.6. Como o align-items mexe com a altura do elemento na box e cada box do elemento acaba sendo do tamanho dele mesmo, não vemos diferença nos valores colocados. A menos, é claro, que alteremos o tamanho da altura da box dos elementos.

3.2.7. Começamos agora a ver as propriedades do grid:

3.2.8. Grid-template-columns: Determina a quantidade e tamanho das colunas do grid. Portanto, se queremos 3 colunas de 100px, escrevemos 100px por 3x:

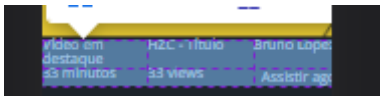
```
.cartao__conteudo {
  display: grid;
  grid-template-columns: 100px 100px 100px;
}
```



3.2.8.1. Ele aceita unidades de medidas como: px, %, em... E a exclusiva **fr**, sendo essa a fração.

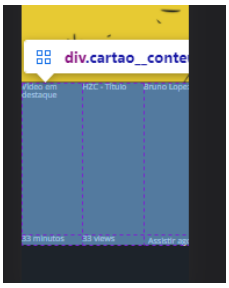
3.2.8.2. A vantagem do fr é a possibilidade de não precisar ficar fazendo contas e trabalhar com valores imprecisos. Se quiser dividir o grid em 3 colunas iguais, não tem como pois 100%/3 é 33.333333.....%, e é aí que entram as frações. Se quiser 3 colunas iguais no grid, basta colocar 1fr por 3x:

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



3.2.9. Grid-template-rows: A mesma ideia do columns, mas para as linhas. Ele inclusive divide o conteúdo da grid automaticamente dentro das colunas caso coloque apenas uma linha:

```
.cartao__conteudo {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: 300px;
}
```



3.2.9.1. Se quisesse que todos os elementos ficassem na mesma linha, não usaríamos grid, mas sim flex.

3.3. Estilização Básica do Cartão:

3.3.1. Overflow: hidder; Propriedade para esconder tudo o que ultrapasse os limites da borda do elemento que estamos estilizando.

3.4. Montando o Cartão Com Grid:

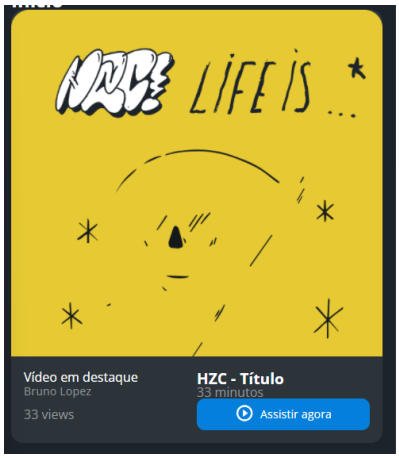
3.4.1. Grid-column: span ***n***.

3.4.1.1. Essa é uma propriedade específica do grid que permite com que você diga ao CSS quantas colunas/células (comparando com excel) você quer que esse elemento específico ocupe (o famoso mesclar no excel. Uso com todos os títulos de planilhas).

3.4.1.2. O valor de N é justamente a quantidade de células queremos que esse elemento específico ocupe.

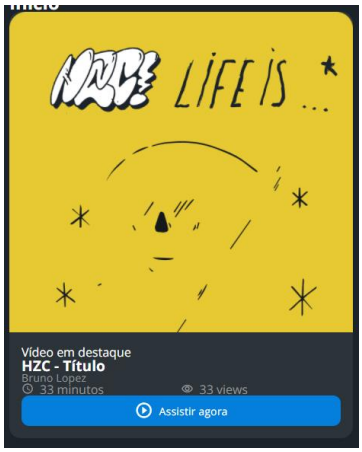
3.4.1.3. Como com o nosso card queremos que somente os minutos e os views fiquem um do lado do outro, mas que todo o resto ocupe a linha inteira, precisamos usar essa propriedade para todos os elementos do nosso cartão, exceto título, perfil e botão, ou seja, o seguinte código será feito para todas as classes referentes a eles:

Sem o grid-column:




```
.cartao__destaque {
  grid-column: span 2;
}
```

Resultado final com o grid-column:



3.4.1.4. Aproveitei e coloquei os pseudo elementos que tinha esquecido. Por isso os ícones de relógio e olho apareceram.

3.5. O que aprendemos:

3.5.1. As limitações de trabalhar com flexbox:

3.5.1.1. A principal delas é trabalhar com dois eixos ao mesmo tempo, eixo vertical e horizontal.

3.5.2. O funcionamento básico do grid:

3.5.2.1. A ideia de grid container é bem parecida com flex container, mas no grid container o fluxo é vertical e também ganhamos acesso a outras propriedades.

3.5.3. Propriedades para criar linhas e colunas: grid-template-rows e grid-template-columns:

3.5.3.1. Os valores que essas propriedades recebem são os tamanhos das colunas/linhas. Ex: para 3 colunas de 30px a propriedade se escreve: grid-template-columns: 30px 30px 30px.

3.5.4. Nova unidade de medida fr:

3.5.4.1. É a unidade de medida para trabalhar com proporções de uma maneira mais simples do que porcentagem. Principalmente quando a porcentagem é uma dízima periódica.

3.5.5. Mescla de linhas e colunas com as propriedades grid-columns: span n e grid-rows: span n:

3.5.5.1. É o conceito de "mescla de células". Serve para dizer quantas colunas/linhas um elemento ocupa dentro do grid container.

4. Aula 4 – Transformando o Layout Com Grid:

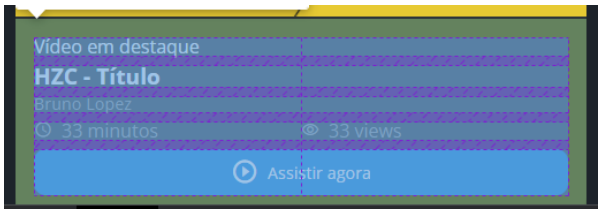
4.1. Avançando Com o Grid:

4.1.1. Gap: propriedade do grip que permite adicionar um espaçamento entre as células/elementos dentro do grid container através de unidades como px, em, rem....:

```
.cartao__conteudo {
  padding: 16px;
  display: grid;
  gap: 8px;
  grid-template-columns: 1fr 1fr;
}
```

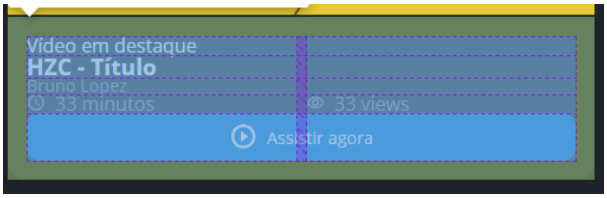
4.1.2. Row-gap: Mesma função que acima, mas somente entre linhas, colunas não se aplicam:

```css
.cartao__conteudo {
  padding: 16px;
  display: grid;
  row-gap: 8px;
  grid-template-columns: 1fr 1fr;
}
```



4.1.3. Column-gap: Exatamente o acima, mas para colunas:

```css
.cartao__conteudo {
  padding: 16px;
  display: grid;
  column-gap: 8px;
  grid-template-columns: 1fr 1fr;
}
```



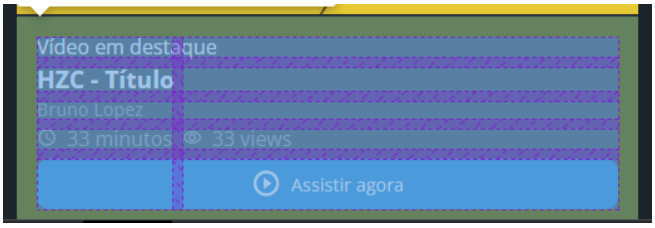
4.1.4. Em uma situação em que o elemento, como no nosso caso de quantos minutos esse vídeo possuí, for subjetivo, podemos utilizar o gird-template-columns: auto 1fr; dessa forma o tamanho da primeira coluna irá sempre se ajustar de acordo com o maior elemento, enquanto a segunda pegará todo o resto do tamanho disponível, uma vez que está ocupando 1 fração.

4.1.5. Utilizamos essa técnica quando temos que ajustar ainda mais o layout. No nosso caso a distância entre os minutos e as views é de 8px no projeto inicial, mas se deixarmos a primeira coluna como 1 fr, e ainda colocarmos um gap de 8px a distância entre ambos acabará sendo muito maior.

4.1.6. Para corrigir isso deixamos a primeira coluna como auto para que ela se ajuste de acordo com o tamanho do maior, e nesse caso único, elemento da nossa primeira coluna, sendo ele os minutos, e então

um gap de 8px tanto para as linhas quanto colunas, ficando exatamente igual o projeto inicial sugere:

```
.cartao__conteudo {
  padding: 16px;
  display: grid;
  gap: 8px;
  grid-template-columns: auto 1fr;
}
```



4.1.6.1. Como dissemos para o grid que todos os outros elementos dele teriam um span de 2 células, nada além dos minutos e views será alterado.

4.2. Visualizando o Grid:

4.2.1. Podemos utilizar uma ferramenta de desenho para tirar um print do nosso layout modelo e desenhar o grid para ter uma noção de como ficará nosso css para atingir aquela expectativa.

4.3. Terminando o Layout da Página:

4.3.1. Criamos um grid no main para organizar melhor as coisas inclusive na versão desktop do site.

4.3.2. Copiamos o cartão destaque e removemos tudo do destaque dele, deixando como um cartão comum.

4.3.3. Nos cartões simples o botão de play fica do outro lado da tela e, no nosso caso, ele estava grudado. Para mudar isso podemos usar o justify-self: flex-end. Como nossos botões são flexbox por conta do pseudo-elemento ícone, podemos fazer essa manobra e manda-lo para o outro lado:

Antes:

```
.cartao__botao {
  align-items: center;
  background-color: #0480DC;
  border-radius: 10px;
```

```
  display: flex;
  font-size: 0.9rem;
  height: 40px;
  justify-content: center;
  width: 40px;
}
```

Depois:

```
.cartao__botao {
  align-items: center;
  background-color: #0480DC;
  border-radius: 10px;
  display: flex;
  font-size: 0.9rem;
  height: 40px;
  justify-content: center;
  justify-self: flex-end;
  width: 40px;
}
```

4.4. O que aprendemos:

4.4.1. As propriedades column-gap, row-gap e gap:

4.4.1.1. São as propriedades que dão espaçamento entre os grid items.

4.4.2. Como utilizar o valor auto para tamanho de colunas:

4.4.2.1. Nem sempre queremos colocar um valor fixo para as colunas/linhas. O valor auto permite que elas se adaptem de acordo com o conteúdo.

4.4.3. Planejar o uso de grid no desenvolvimento:

4.4.3.1. Uma técnica muito interessante é usar alguma ferramenta de desenho e esboçar possíveis linhas e colunas em cima do layout recebido.

5. Aula 5 – Responsividade Com Grid:

5.1. Quebrando O Projeto No Desktop:

5.1.1. Fizemos a quebra do projeto desenhando como será feito nosso grid através do modelo.

5.2. Adaptando A Estrutura Para Desktop:

5.2.1. Adicionamos o cartão de vídeos recentes com um article, um h3 e uma lista com todos os vídeos recentes:

```
<article class="cartao cartao--recentes">
      <h3 class="cartao__titulo">Videos recentes</h3>
      <a href="#" class="cartao__link">Ver todos</a>
      <ul class="cartao__lista">
        <li class="cartao__item">
          <img src="assets/img/miniatura_1.png" alt="Thumbnail"
class="cartao__item-thumbnail">
          <h4 class="cartao__item-titulo">HZC - Love machine</h4>
          <p class="cartao__item-perfil">Bruno Lopez</p>
        </li>
</article>
```

5.2.1.1. Essa foi a estrutura principal, o li foi copiado 5x e esse article copiado para antes do segundo cartão de destaque la em baixo.

5.2.1.2. Tudo está sendo feito para alcançar o layout idealizado pelo modelo no figma.

5.3. Estilizando o Cabeçalho com Grid:

5.3.1. Quando usamos grid podemos usar a função repeat(x, n) como valor para todas as propriedades. X é a quantidade de vezes que o N (valor) irá ser repetido:

```
grid-template-columns: repeat(3, auto);
```

5.4. Criando As Colunas Da Classe Principal:

5.4.1. Criamos um media query para o menu lateral trazendo ele de volta a pág com o position: static. E um outro para o main colocando ele como um grid de 3 colunas onde a primeira e última tem tamanho automático para ficarem do mesmo tamanho do conteúdo, ou seja, auto, e a do meio com o restante do espaço, ou seja, 1fr.

5.5. Organizando Elementos Com grid-row e grid-column:

5.5.1. Podemos dizer quais serão as linhas e colunas que cada elemento ocupará utilizando as propriedades acima, ou seja, não servem apenas para o span que é para mesclar as células.

5.5.2. Se quisermos tanto indicar antas linhas e colunas nosso elemento irá ocupar e ainda mesclar células, podemos usar os dois valores juntos separando por '/':

```
grid-row: 1 / span 2;
```

5.6. O que aprendemos:

5.6.1. Como evoluir o layout para desktop com grid:

5.6.1.1. Como fica o planejamento e criação das colunas e linhas da página quando existe um espaço horizontal maior em um dispositivo desktop.

5.6.2. A função repeat():

5.6.2.1. Quando precisamos criar muitas colunas com o mesmo tamanho, evitando repetição de código.

5.6.3. Alterar a posição de elementos com as propriedades grid-row e grid-column:

5.6.3.1. Antes usadas apenas para mesclar linhas e colunas, vimos que essas propriedades também controlam onde um elemento começa e termina dentro do grid container.

6. Aula 6 – Grid Container e Suas Áreas:

6.1. Manutenção do Código:

6.1.1. Grid-template-area: Ao invés de linhas e colunas como o grid-columns/rows, essa propriedade designa áreas para determinados títulos e toda a tag que possuir aquele título específico, ocupara a área que foi designada:

```
grid-template-areas: "titulo-pagina titulo-pagina titulo-pagina";
```

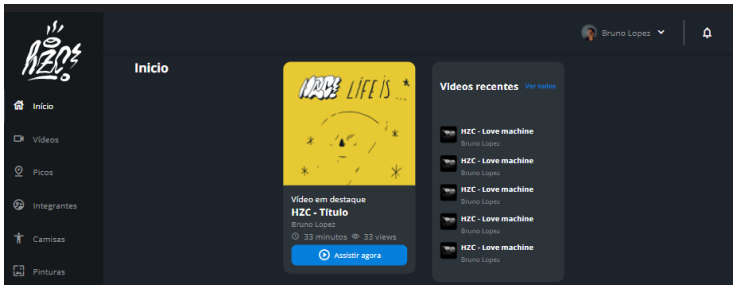
6.1.1.1. Escrevemos 3 vezes pois estamos determinando que nossa área terá 3 colunas e que essa primeira linha será inteira do titulo-template.

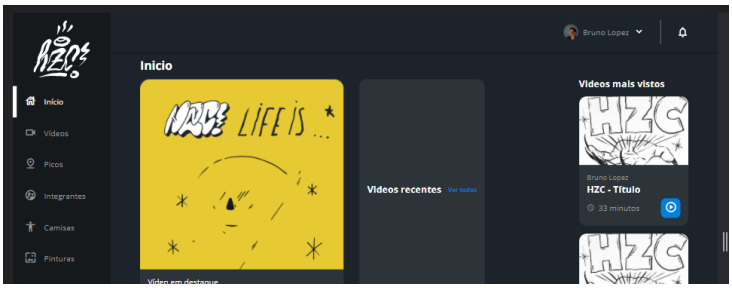
6.1.1.2. Essa propriedade foi usada no grid container do main.

6.1.1.3. Agora vamos à tag h2 que queremos que ocupe esse espaço e utilizamos o grid-area nele nomeando como titulo-pagina:

```
grid-area: titulo-pagina;
```

6.1.1.4. Resultado antes e depois:





6.1.2. Se quisermos criar uma segunda linha, basta dar 1 espaço e abrir outras “” nomeando as áreas novamente:

No main:

```
grid-template-areas: "titulo-pagina titulo-pagina titulo-pagina"
"destaque destaque recentes";
```

No cartão—destaque e –recentes:

```
grid-area: destaque;
grid-area: recentes;
```

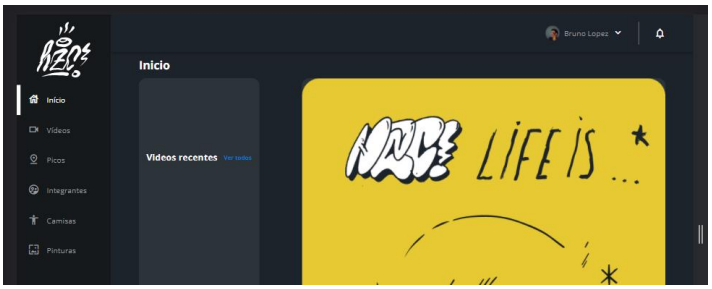
Resultado:

6.1.3. A vantagem de usar áreas é poder inverter elas no grid container e tudo automaticamente ser trocado de posição:

No main:

```
grid-template-areas: "titulo-pagina titulo-pagina titulo-pagina"
"recentes destaque destaque";
```

Resultado:



6.1.4. Essa técnica é super vantajosa e prática para futuras alterações.

6.2. Grid Áreas:

6.2.1. Para melhorar a leitura do nosso código podemos escrever nosso template da seguinte forma:

```
grid-template-areas:
        "col1 col2 col3"
        "col1 col2 col3"
        "col1 col2 col3"
    ;
```

6.2.2. Além disso podemos criar classes específicas para cada elemento que dará match nessas áreas com o nome que será dado para cada área, dessa forma essa classe servirá unicamente para o template:

```
.destaque-video {
  grid-area: destaque-video;
}

.videos-recentes {
  grid-area: videos-recentes;
}
```

6.3. O que aprendemos:

6.3.1. As vantagens e utilização de grid áreas:

6.3.1.1. Grid areas vem com o propósito de facilitar a manutenção de código e a visibilidade dos elementos dentro do grid container.