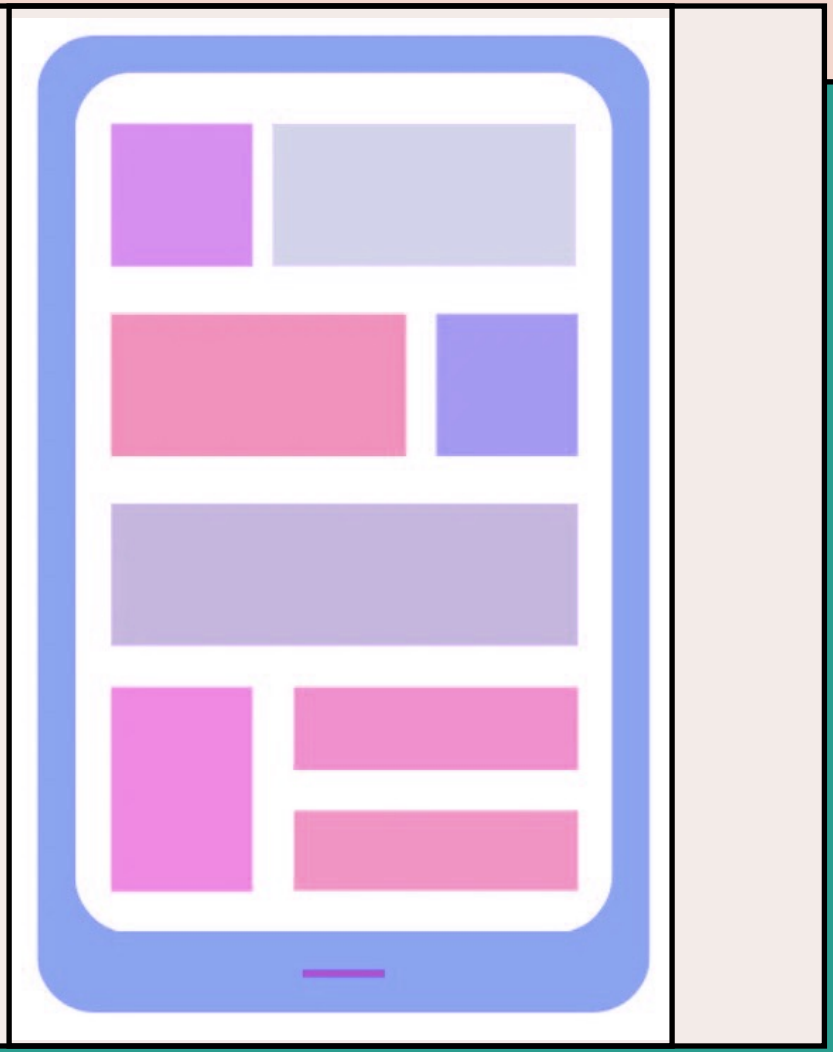
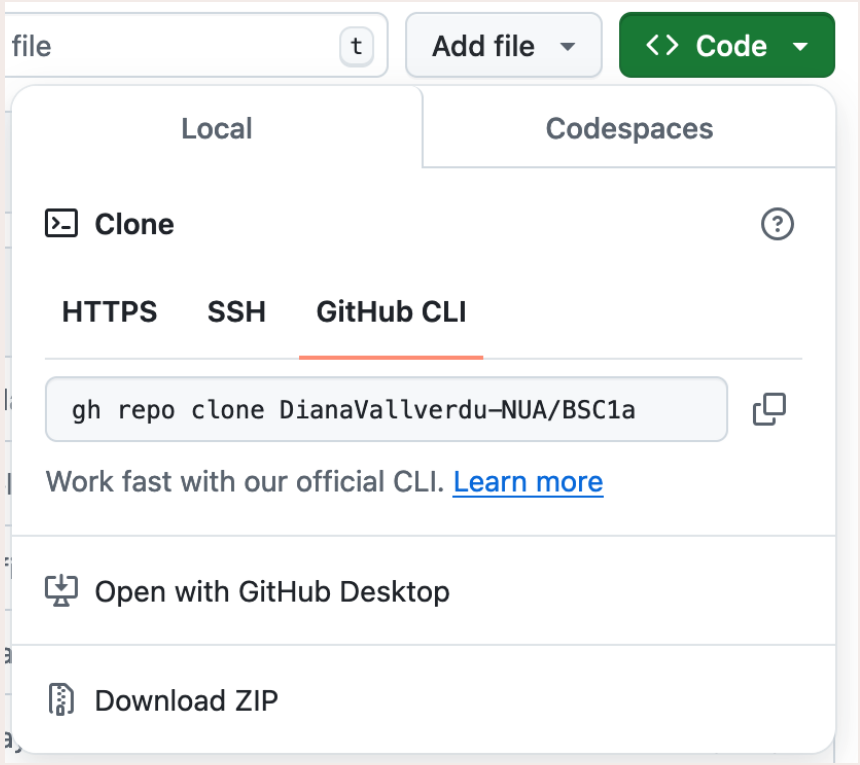
CSS: Layouts

Hannah-Louise Batt

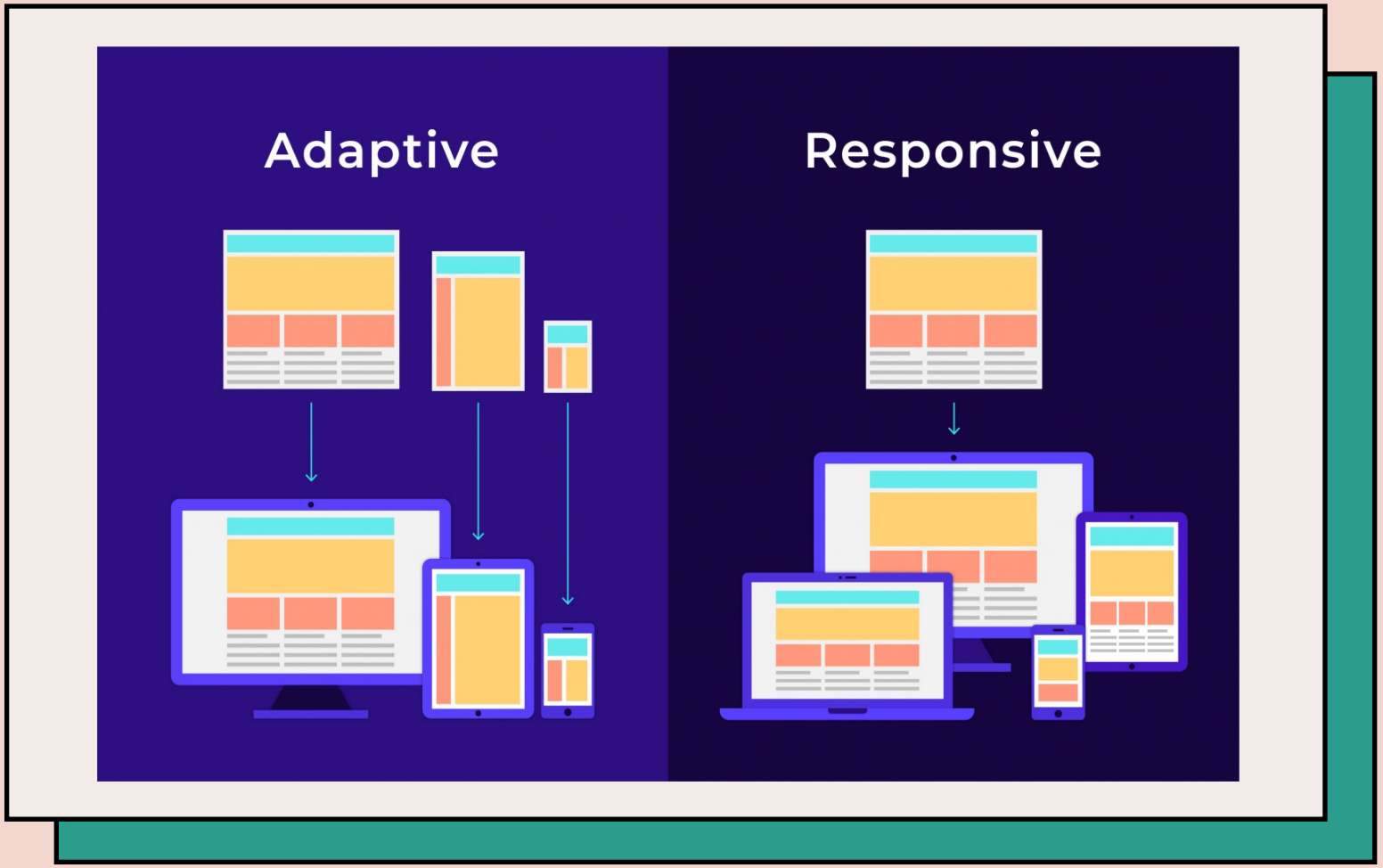


Exercise project folders

- Go to https://github.com/DianaVallverdu-NUA/BSC1a/tree/technical-workshops
- Download the project folder by going to **Code -> Download ZIP**, or by cloning the repository with Github
- Open the folder in Visual Studio Code
- We'll be working in the sub-folder **04 CSS Layouts** today, but feel free to add your own styles and customise your folder as we go!



What is Responsive Web Design?



- **Responsive Web Design (RWD)** is an approach to web design that centres on the idea that web pages should look the same across screen sizes, resolutions and devices.
- This practice makes use of fluid layout styling and media queries, to make content responsive to the **Viewport** it's viewed in.
- It's beneficial to have this principle in mind when styling your pages.

Normal Flow

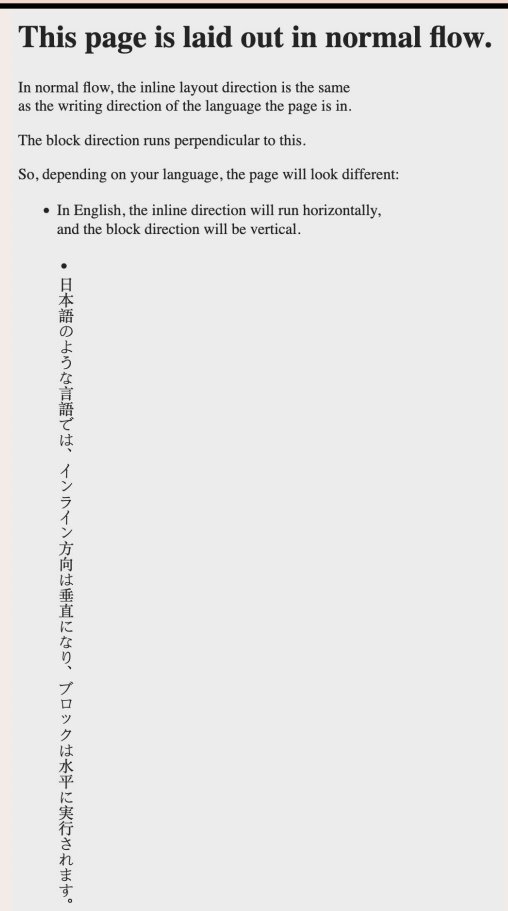
```
<h1>This page is laid out in normal flow.</h1>

<p>In normal flow, the inline layout direction is the same <br/>
    as the writing direction of the language the page is in.</p>

<p>The block direction runs perpendicular to this.</p>

<p>So, depending on your language, the page will look
different:</p>

<ul>
  <li>In English, the inline direction will run horizontally, <br/>
  and the block direction will be vertical.</li>
  <br/>
  <br/>
  <li style="writing-mode: vertical-rl;">日本語のよ
  な言語では、インライン方向は垂直になり、
  ブロックは水平に実行されます。</li>
</ul>
```



- **Normal Flow** is what happens when we make no changes to a HTML page's layout.
- HTML elements are displayed in the exact order they appear.
- We've used **block** elements, so our elements are displayed stacked on top of one another.

Just remember that the Western-style layout we're used to, whilst increasingly popular, isn't the only kind that exists!

私たちは皆さまに寄り添い、
地域のため、社会のために
力を尽くしておられる方々を
精一杯応援する法律事務所です。

当事務所は、地域のため、社会のため、公共のために、日々、頑張っておられる皆さまを応援する法律事務所です。
地域や社会が今よりもっと良くなるように、美しい自然や季節の風景を後世に引き継げるように、社会の一人ひとりが今よりもっと幸せに暮らせるように、そして、大切な人や家族が幸せでいられるようにと、頑張っておられる皆さまに寄り添い、応援します。
地域の大事な財産を守り、人や自然にやさしい未来を創るために頑張ります。

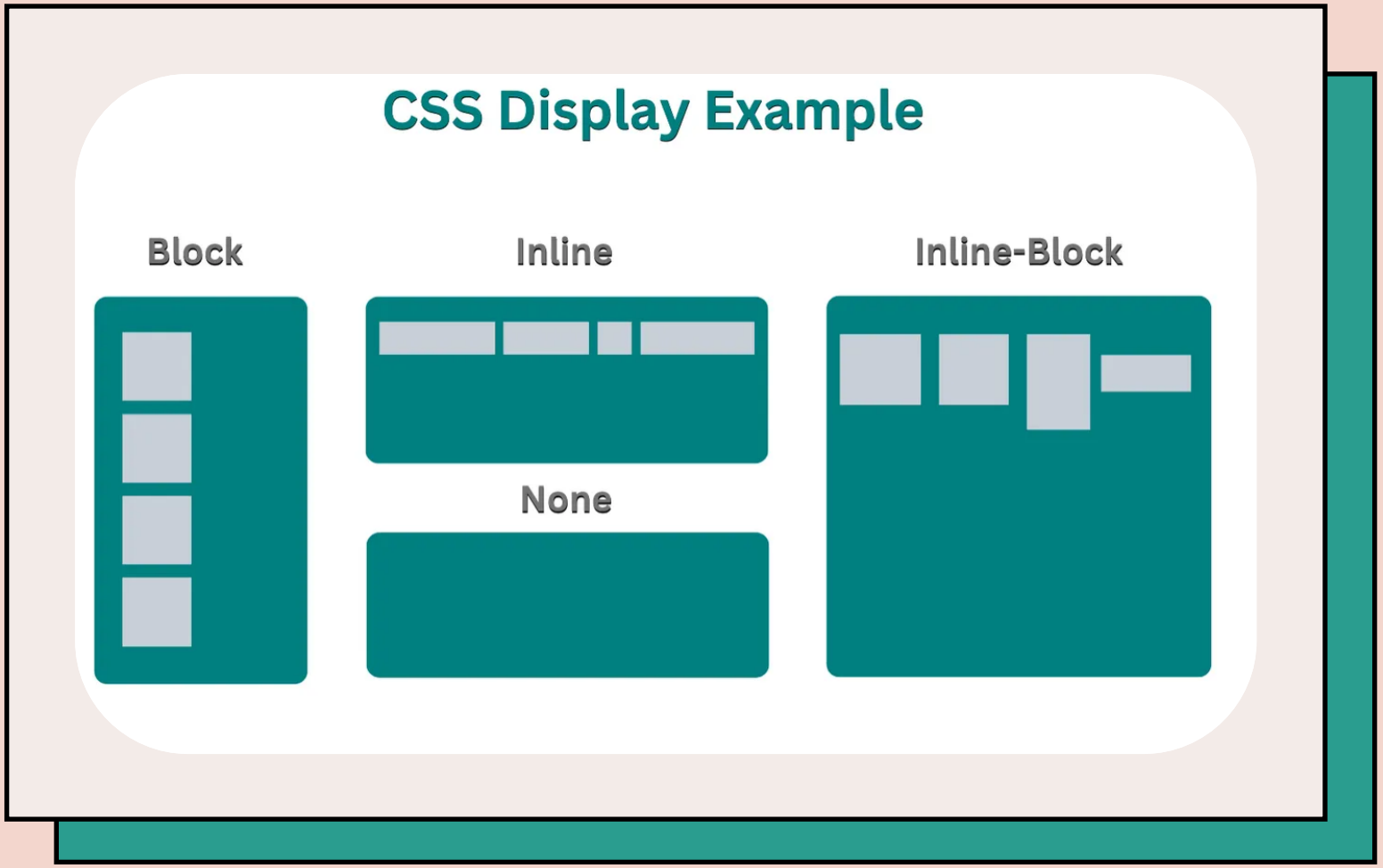
おかやま丸の内法律事務所

Okayama Marunouchi Law Office

メッセージ
弁護士プロフィール
業務のご案内
アクセス
ロゴについて

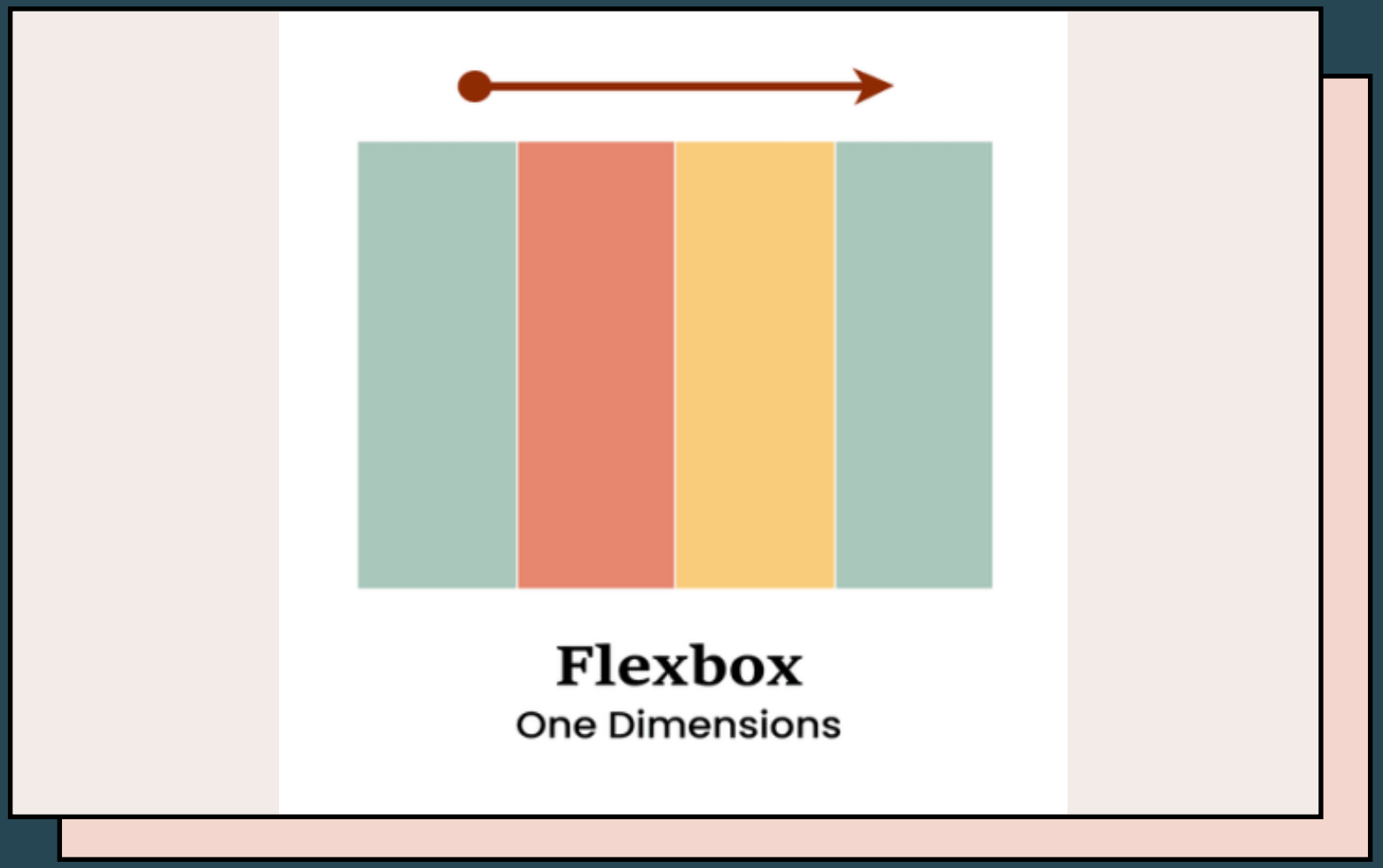
〒700-0822 岡山市北区表町1-5-1岡山シンフォニービル1F
営業時間 9:00-17:00（土日祝休み）
Tel 086-238-2525 Fax 086-238-5454

The Display Property: Revisited



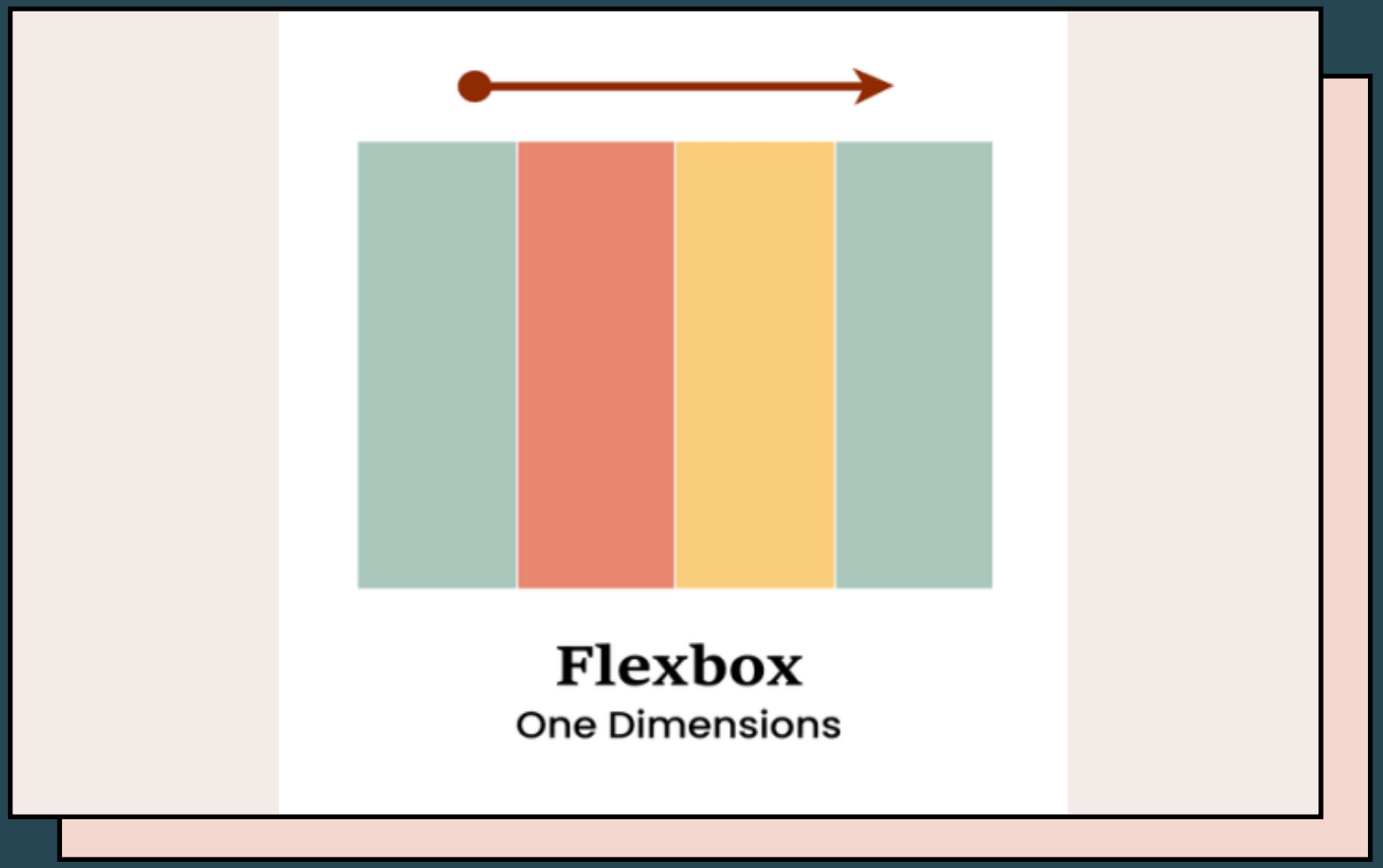
- Our main method for crafting different layouts in CSS is by using the **display** property.
- By using **block, inline** or **inline-block** values, we can change how elements behave in normal flow.
- However, these aren't the only values we can give the display property. We can activate entirely new layout methods by using two new values: **flex** and **grid.**

Why would I use Flexbox?



- **Flexible box layout (Flexbox)** allows us to arrange our elements in rows or columns.
- Items inside the flexbox will expand evenly to fill extra space, or shrink to fit smaller areas.
- We call Flexbox a **one-dimensional layout**, because it only deals with a single dimension at a time – either a row or a column.

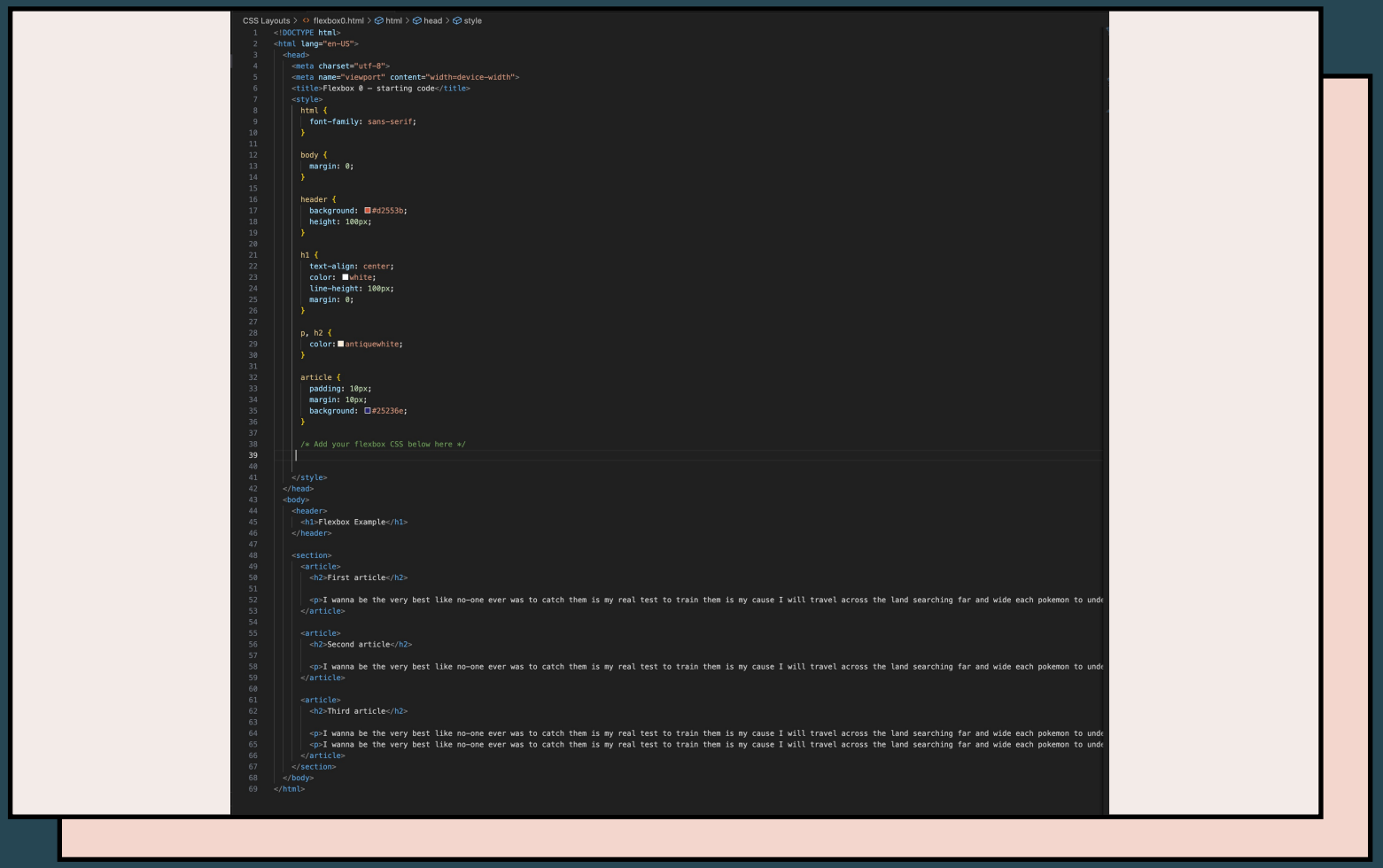
Why would I use Flexbox?



It's useful to use Flexbox when we want to:

- Vertically centre our content inside its containing element
- Ensure that all elements inside a container take up an equal amount of space on the page, no matter the viewport size
- Create multi-column layouts with identical column heights, even when they have different amounts of content.

Flexbox: Example Activity



Let's take a look at a practical example.

- In your VSCode project folder for these sessions there's a HTML file called **flexbox0**.
- In this file you'll see some CSS styling in the <head>, along with a <header>, and a <section> element with three nested <article> elements.
- As article elements are **block** elements, how do we think these will look on the webpage in normal flow?

Flexbox: Example Activity

```
section {
    display: flex;
}
```

- Now, if we want our articles to be evenly spaced out, we need to adjust the display property of the element that contains them – the <section> element.
- Let's add a section {} selector to the end of our CSS, and create a rule setting the **display** property to **flex.**
- Now save your code **(CTRL/CMD + S)** and refresh the page.
- Note how all three boxes are now the same size, even though the last box contains more text.

Flexbox: Example Activity

```
section {
    display: flex;
    flex-flow: column;
}
```

- If we want our flex items to run in rows instead of columns, like how we had them at the beginning, we can use the **flex-flow** property.
- Setting our **flex-flow** to **column** will cause the blocks to be stacked vertically; **row** will lay them out horizontally.
- Try adding **flex-flow** into your CSS and see the change!

Flexbox: Example Activity

```
article {
    padding: 10px;
    margin: 10px;
    background: #25236e;
    flex: 200px;
}
section {
    display: flex;
    flex-flow: row wrap;
}
```

We're also adding a new property into our article {} selector, which sets a minimum size each flex item can be.

- Now, take a look at the **flexbox1** HTML file and how it displays in the browser. It looks a little busy, right?
- Flexbox will by default try to place all its flex items evenly onto a single line – which can lead to a very crowded looking page!
- We can fix this by adding another value, **wrap**, to our **flex-flow** property.
- **flex-flow** is actually the shorthand for two separate properties: **flex-direction and flex-wrap.**

Flexbox: Flexible Sizing

```
article {
  flex: 1;
}
```

We can also control the size of a flex item relative to the ones around it – creating more interesting layouts.

- Head back to the **flexbox0** file.
- At the bottom of your CSS, add the CSS rule on the left.
- What we've just done is given our articles a proportion value, dictating a fraction of available space they should take up.
- Since all our articles have the same value, nothing on our page should change. Let's fix that.

Flexbox: Flexible Sizing

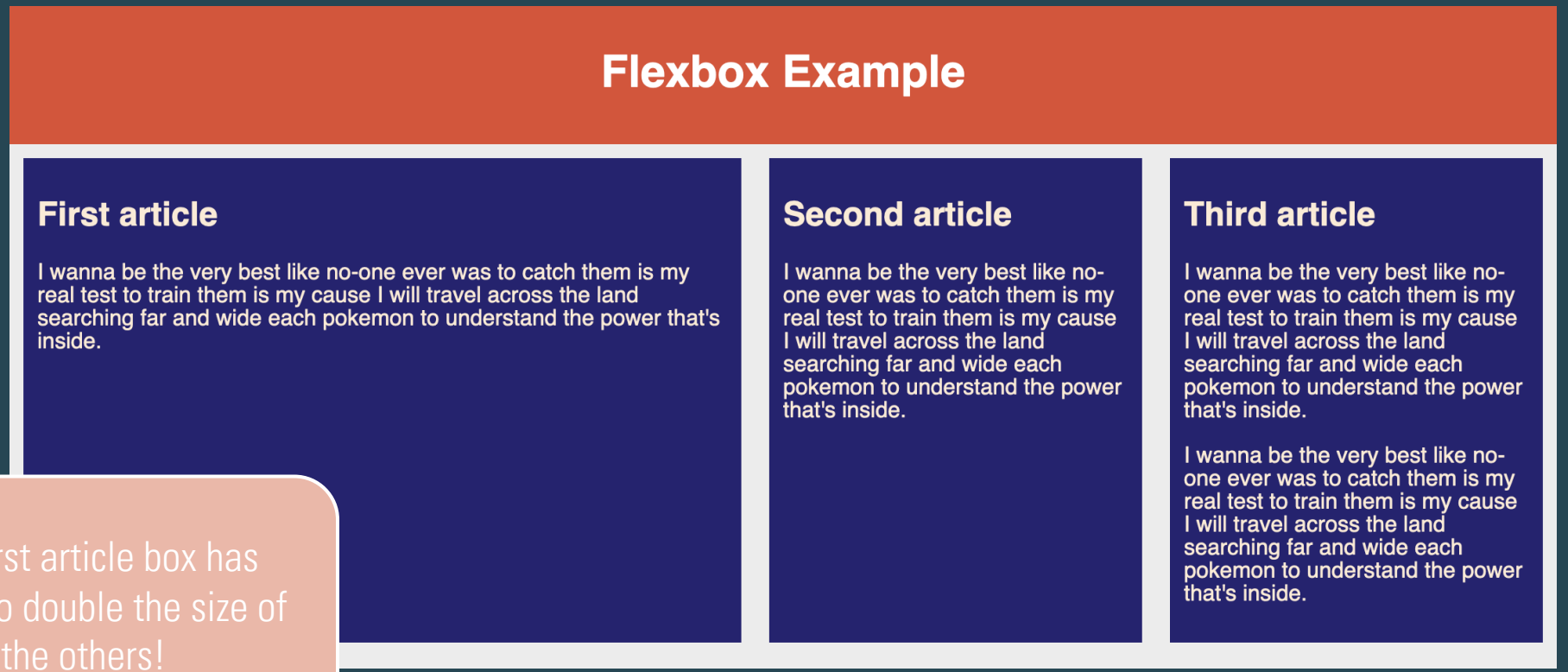
```
article {
  flex: 1;
}
article:nth-of-type(1) {
  flex: 2;
}
```

Now, add this second rule to your CSS.

- This time, we're using a **pseudo-class** to select a specific <article> based on its position amongst other elements of the same type.
- We've chosen the first <article> element on our page and given it a new flex value of 2.

What do we think this will do to our page?

Flexbox: Flexible Sizing



Our first article box has grown to double the size of the others!

The flex Property

The resizing we just did is only one of the several ways we can use the **flex** property. **flex** is shorthand for three separate properties:

flex-grow

Assigns a element with a unitless proportion value for sizing. This is what we just used.

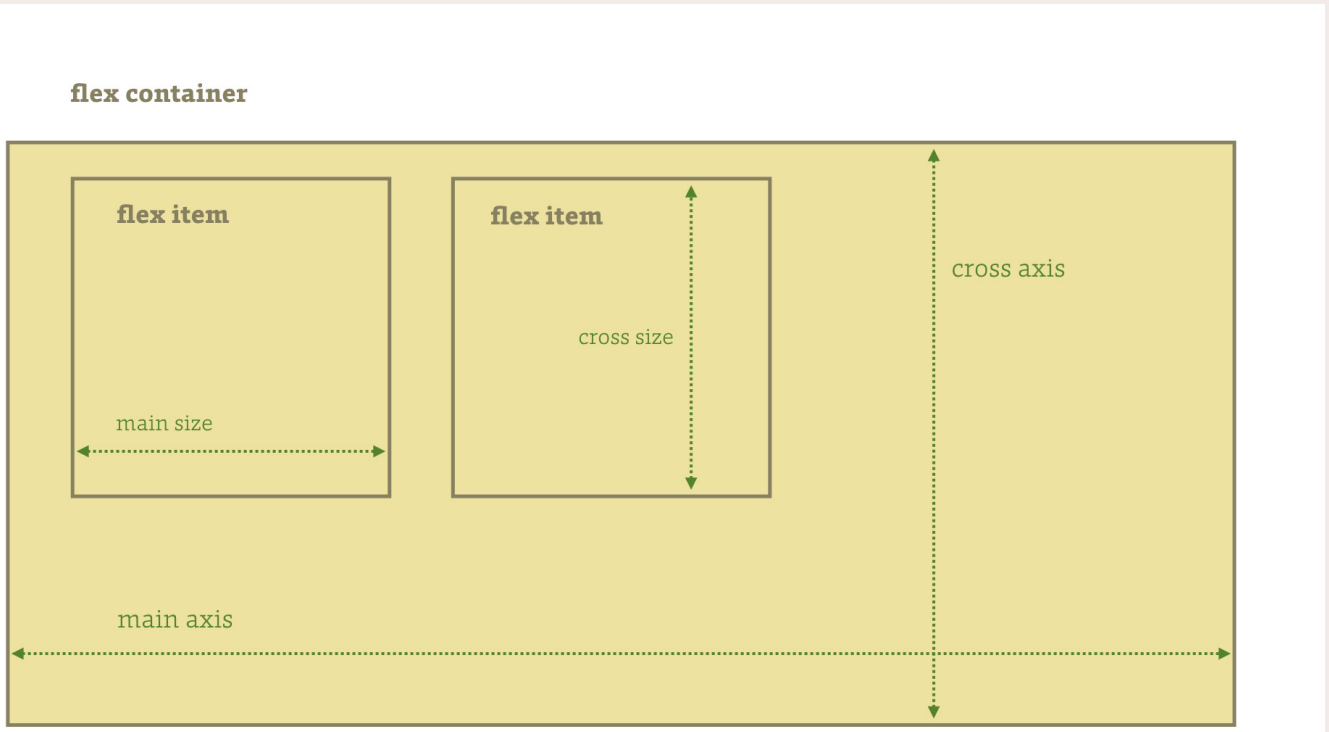
flex-shrink

Another unitless proportion value. Defines how much a flex item will shrink compared to other elements in the event that our flex items are too large for their container.

flex-basis

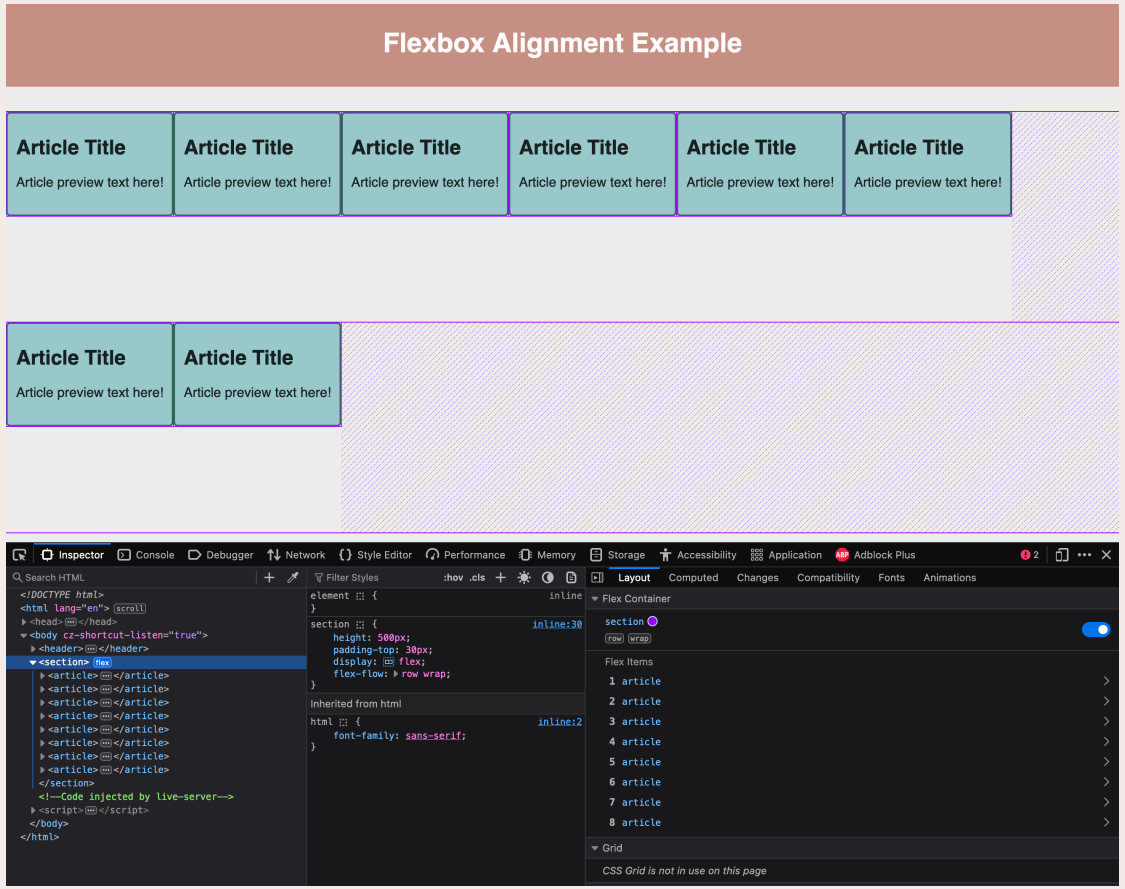
A minimum size value each flex item should have.
When using this *and* **flex-grow,** each flex item will get the allocated space defined in **flex-basis** before being assigned a proportion of what's left.

The Flex Model



- Just like the element box model, there's a model for Flexbox too.
- The **main axis** runs in the direction in which flex items are laid out. Flex items will start from the edge of the main axis by default.
- The **cross axis** runs perpendicular to the main axis. Flex items will stretch to fill this axis by default.
- Everything we do with flexbox runs along these two axes, so it's worth learning how they work if you want to style a page this way.

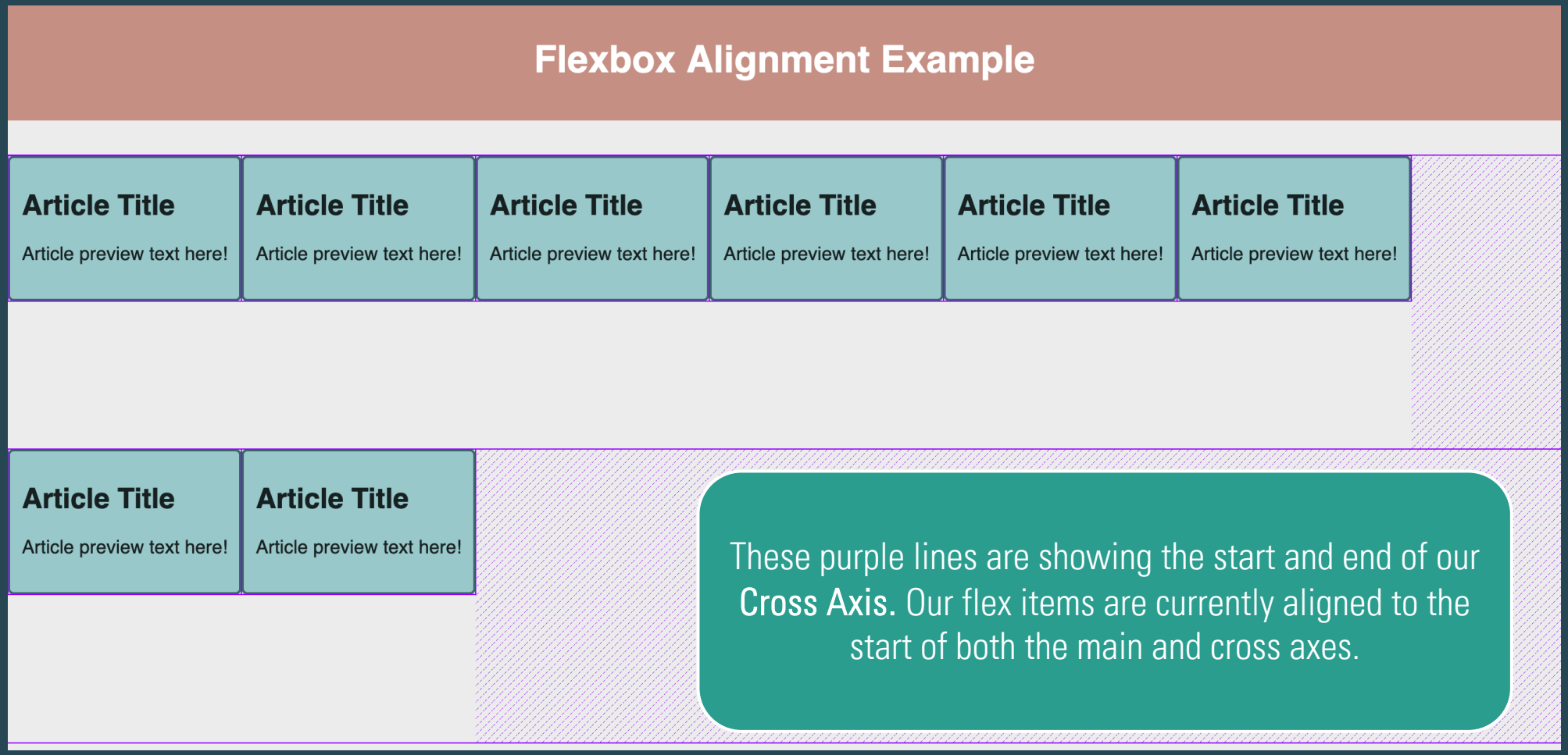
Flexbox: Alignment



Now we have a grasp on the Flex Model, we can think about how we align items in our flexbox along these axes.

- Open up the **flexbox2** file and preview it in the browser.
- Open up your developer tools **(F12)** and select <section> on the left hand side.
- Your browser should have some kind of flexbox preview mode that will allow you to view your flex container size and where your flex items sit within it.

Flexbox: Alignment



Flexbox: Alignment

```
section{
    padding-top: 30px;
    height: 500px;
    display: flex;
    flex-flow: row wrap;
    /* Adding our flexbox alignment rules in here */
   align-items: center;
   justify-content: space-around;
}
```

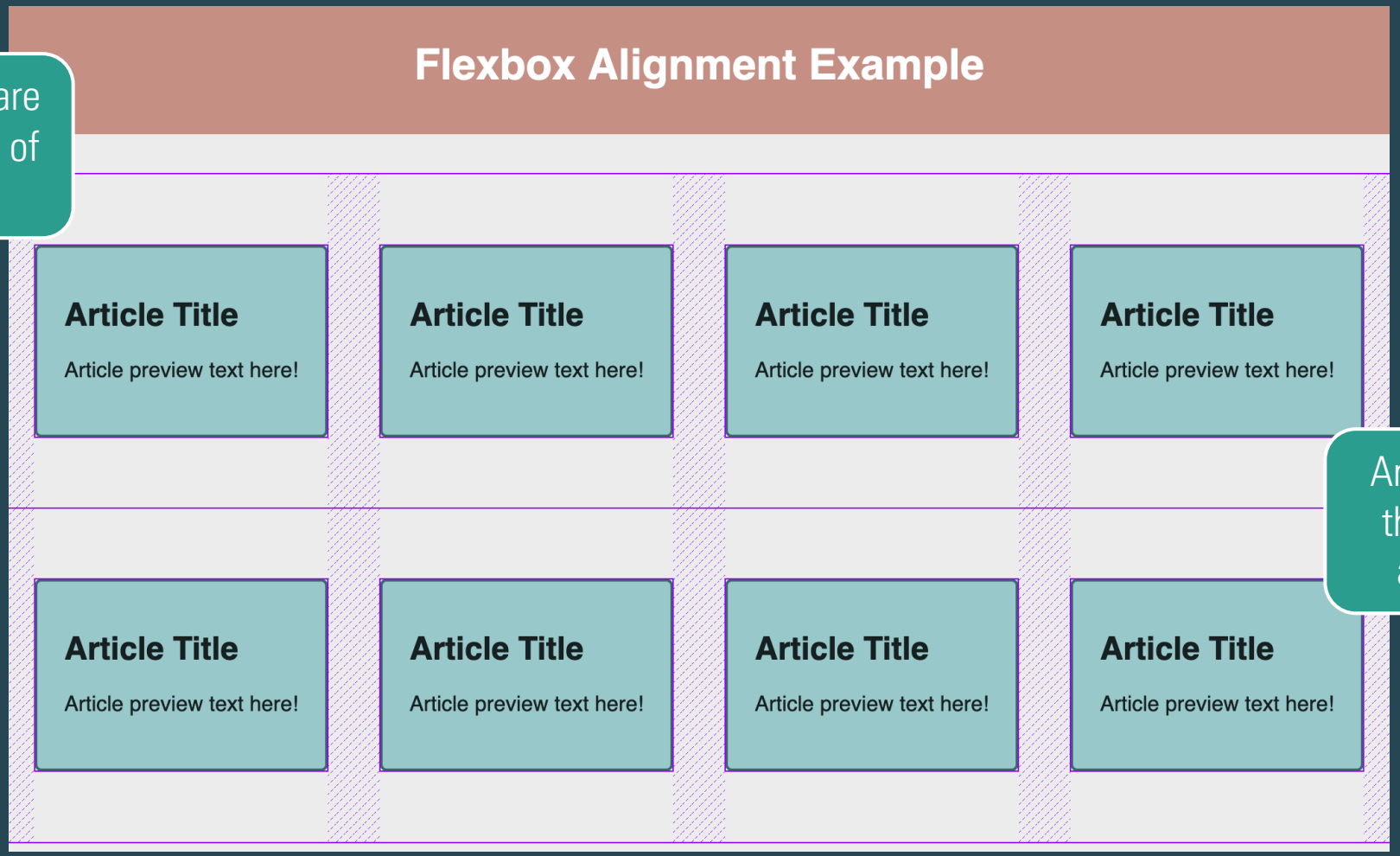
If we want to change the alignment of our flex items, we're going to use two properties: **align-items** and **justify-content.**

- **align-items** will adjust our content along the cross axis
- **justify-content** will space out our flex items along the main axis.

Add these two new rules into your file, one at a time, saving in-between. See how the alignments change!

Flexbox: Alignment

Now, our flex items are aligned to the centre of the cross axis…



And spaced evenly on the page with space around the edges!

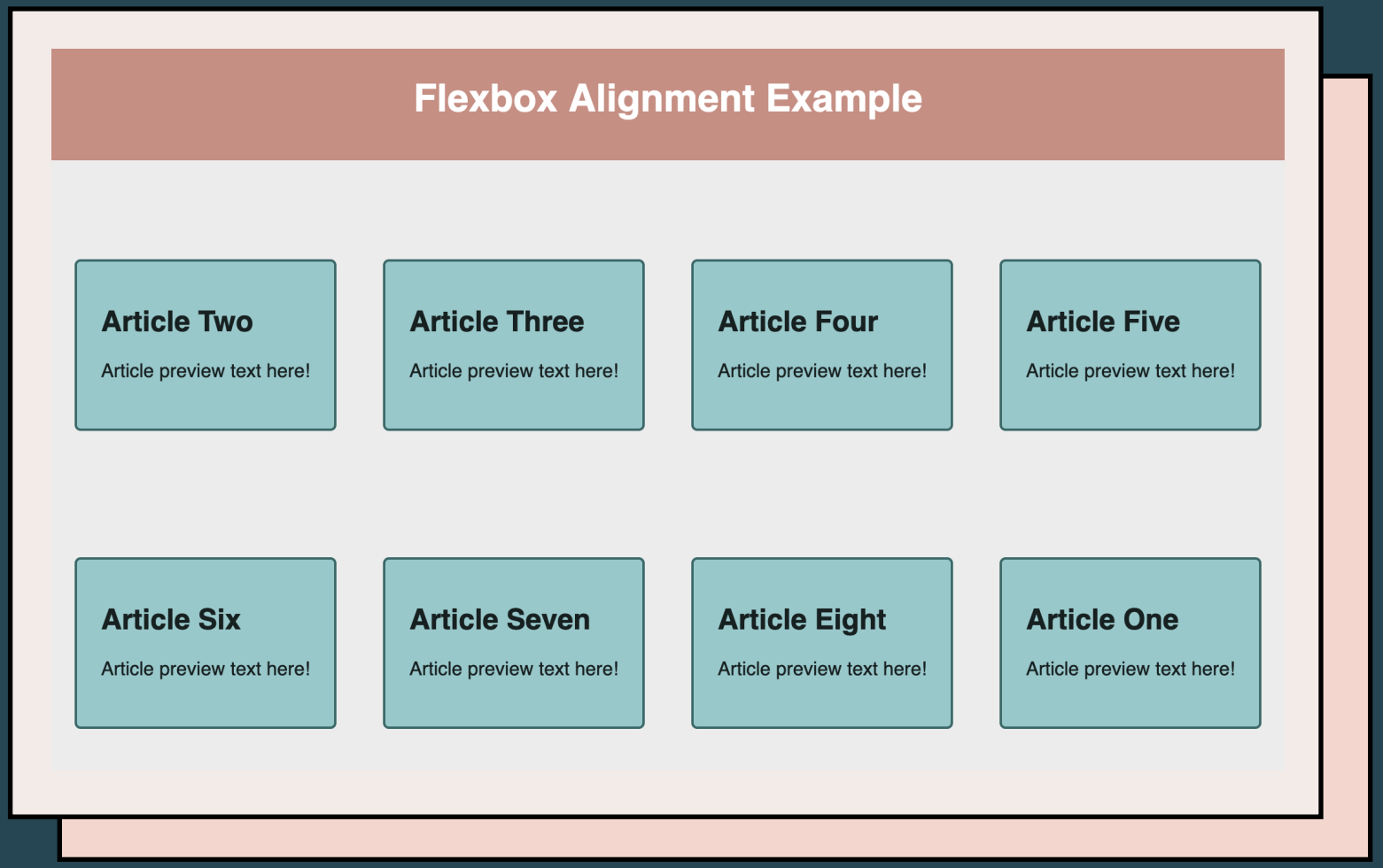
Flexbox: Ordering

```
article:first-child {
    order: 8;
}
```

Another great thing about Flexbox is that we can order our content without messing up our **source order**.

- Back in **flexbox2,** let's create a new CSS rule targeting the first article on our page.
- By default, all flex items have an order value of 0. By increasing that value, our article moves down the order.

Flexbox: Ordering

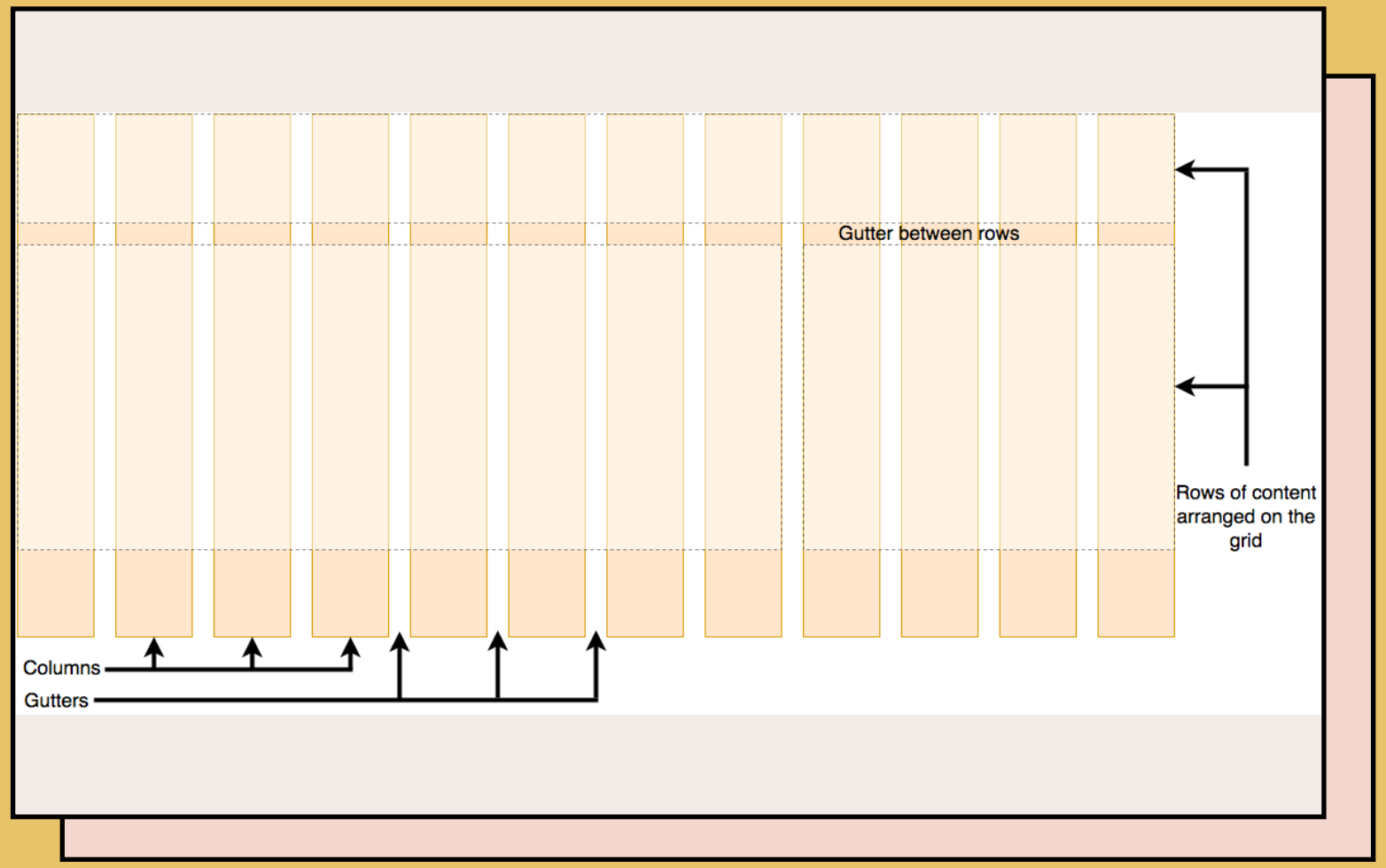


- Our first article has now moved down to last in the order!
- If we want an item to appear earlier in the order, we can use a negative value instead.

Break

Take 10 minutes!

Why would I use CSS Grid?



- CSS grid is a two-dimensional layout system, letting us organise content into columns *and* rows.
- Grid is used to create elements that won't resize or shift on multi-page sites, so it's great for maintaining consistency.
- A grid will be comprised of rows, columns, and a third value: the gaps between those columns and rows. These are referred to as **gutters.**

Creating a CSS Grid

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr) 500px;
    gap: 15px;
}
```

Let's play with another example:

- This time, find the **grid0.html** file in VSCode.
- If you open this file in your browser, the containers are currently laid out in normal flow.
- We can also see a parent element with a class - **.container.** Let's use that class as our selector to create the grid.
- We're adding in 3 rules into our declaration. Now save and refresh!

Creating a CSS Grid

Let's break down a bit more what you're seeing here:

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr) 500px;
    gap: 15px;
}
```

We're using the display property to set the layout to **grid.** On its own, through, this doesn't do a whole lot.

Creating a CSS Grid

Let's break down a bit more what you're seeing here:

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr) 500px;
    gap: 15px;
}
```

It's the **grid-template-columns** property doing the majority of the heavy lifting in creating our grid. This property explicitly defines that we want to create a certain number of columns.

Creating a CSS Grid

Let's break down a bit more what you're seeing here:

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr) 500px;
    gap: 15px;
}
```

We then have two types of values we've given: a **repeat ()** function, and a single pixel value of 500px.
With this, we can already identify that our last column will be 500px wide.

Creating a CSS Grid

Let's break down a bit more what you're seeing here:

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr) 500px;
    gap: 15px;
}
```

The **repeat ()** function allows us to create a batch number of columns which are all the same size. Here, we've created 3.

Creating a CSS Grid

Let's break down a bit more what you're seeing here:

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr) 500px;
    gap: 15px;
}
```

The second parameter in this function, **1fr**, is a **fraction value.** Fraction values divide up available space in the grid's container into equal parts, then give each column the number of fractions declared.

Creating a CSS Grid

Let's break down a bit more what you're seeing here:

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr) 500px;
    gap: 15px;
}
```

It's important to note that the available space for fractions is calculated **after** absolute pixel values are deducted. So if our maximum space is 900px, and the last column is 500px, that leaves 400px to be divided equally between the first 3 columns.

Creating a CSS Grid

Let's break down a bit more what you're seeing here:

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr) 500px;
    gap: 15px;
}
```

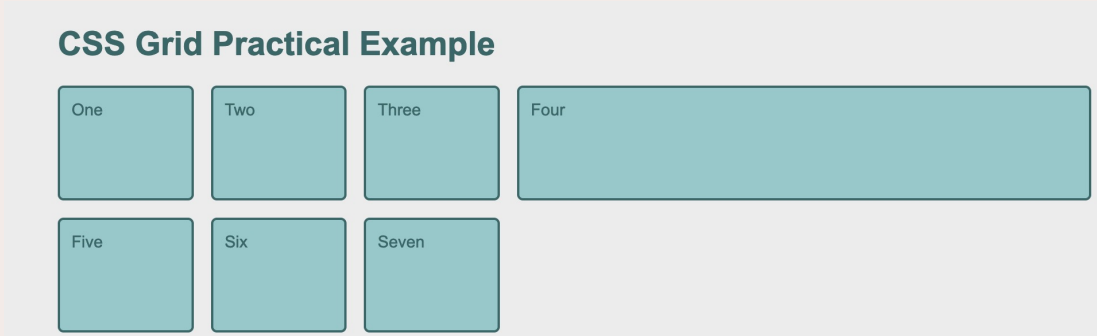
The **gap** property defines the size of the **gutter** between rows and columns. It's shorthand for the **column-gap** and **row-gap** properties: only use it if you want your row and column gutters to be the same size.

Creating a CSS Grid

So, just to recap:

- We have created a 4-column grid in a 900px container
- The last column is 500px wide
- The gap between columns and rows is set to 15px
- That means the first three columns are all ~118px wide.

Implicit & Explicit Grids




```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr) 500px;
    gap: 15px;
    grid-auto-rows: 100px;
}
```

Notice how our example automatically created an extra row for 5-7?

- This is the difference between **Implicit** and **Explicit** grids.
- We declared a set number of 4 columns - an **Explicit** grid.
- But because we have more than 4 containers, an **Implicit** grid is created to expand the rows to fit our content.
- Implicit grids are auto-sized to fit around the content inside them. If you need your rows to be a specific size, you can use the **grid-auto-rows** property to set it.

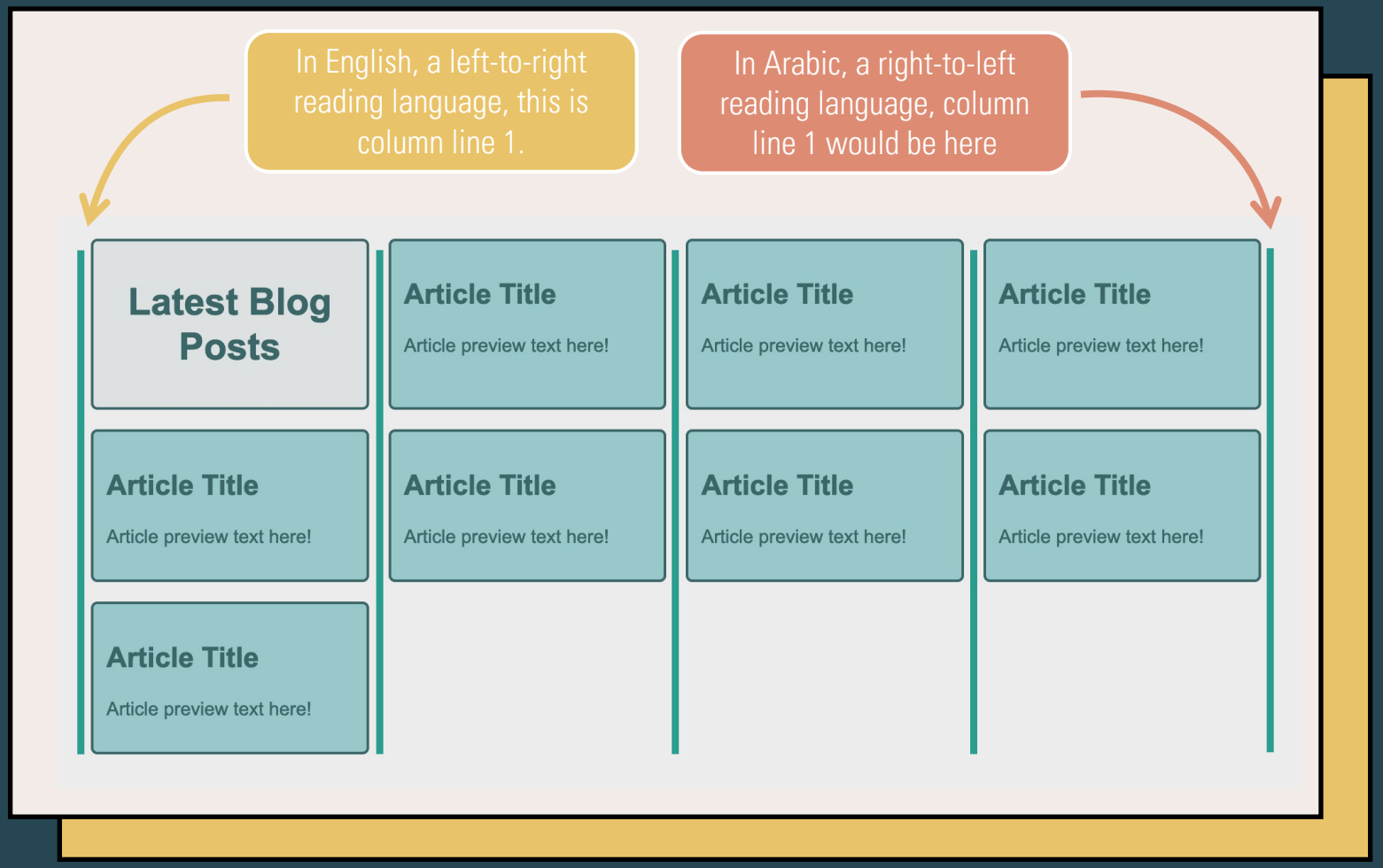
minmax()

```
.container {
    display: grid;
    grid-template-columns: repeat(3, 1fr) 500px;
    gap: 15px;
    grid-auto-rows: minmax(100px, auto);
}
```

adding **auto** as the max value means rows will always grow to fit extra content.

- If we set our tracks to a specific size, that sets a hard limit on what we can fit in our rows before they start to overflow.
- Luckily, we can use the **minmax()** function to set a minimum and a maximum value for our row size.
- This means that our row will always be at least 100px, but can stretch to a larger size if we add more content.

Line-based Placement



Now we have a grid, we can place content on it.

- For line-based placement, envision each row and column as a line.
- We can then specify in CSS where an element starts and ends within the grid, using the **grid-row** and **grid-column** properties.
- Head to the **grid1** file in VSCode and take a look. We've got a header and 8 articles in our grid.
- We want our header to cover the entire first row. Let's look at the CSS rules for the header element.

Line-based Placement

```
header {
        border-radius: 5px;
        padding: 10px;
        background-color: rgb(221, 225, 225);
        border: 2px solid rgb(58, 103, 103);
        text-align: center;
        grid-column: 1 / 5;
        grid-row: 1;
    }
```

grid-column is CSS shorthand, combining the **grid-column-start** and **grid-column-end** properties.

- Our **grid-column** property needs two values: a starting line and an ending line.
- We want to start at the beginning of the first column, so we'll put line 1 as our first value.
- So that our header covers the fourth column, we're going to end our placement at line 5. We separate these values with a slash.
- For **grid-row,** we only need to tell it to start at line 1, so we only add one value.

Line-based Placement

Much better!

Line-based Placement

What would we need to do to the first article element to get a layout like this?

Line-based Placement

In the **grid2** file, have a play about with the placement values in the **.bigBox** class and see how it changes your page layout!

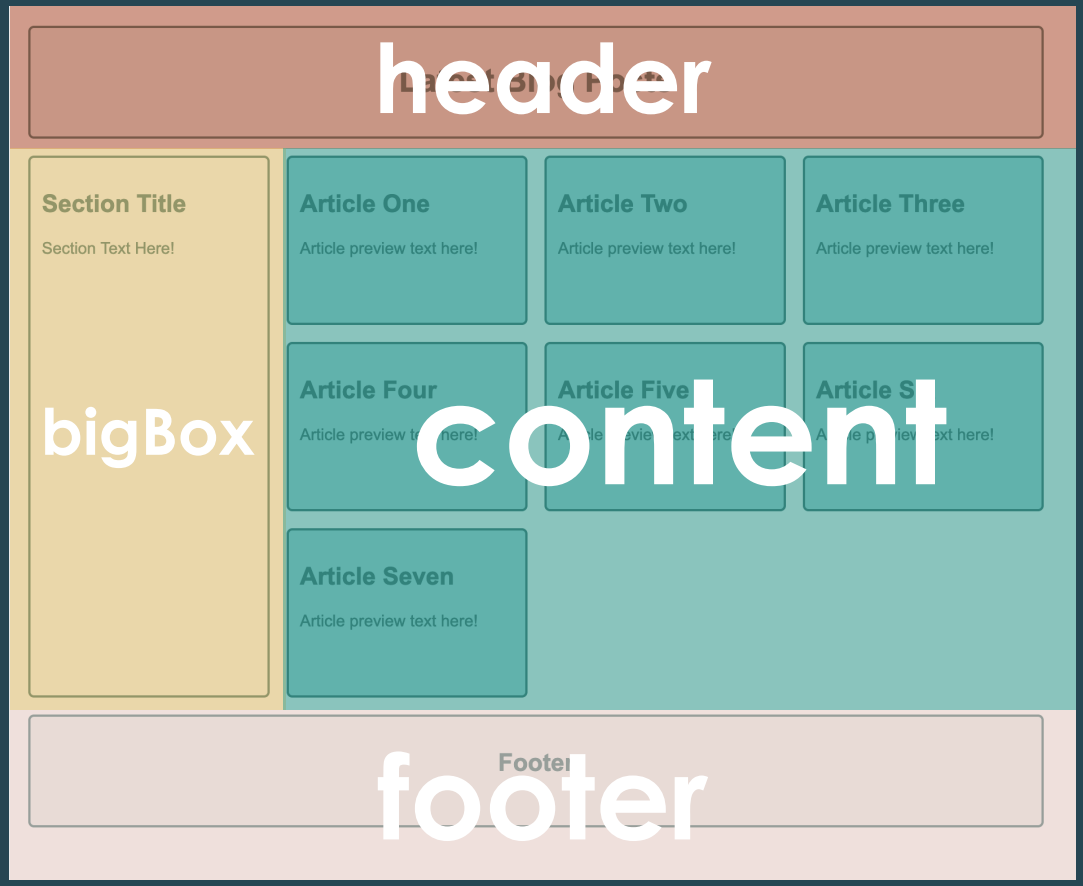
grid-template-areas

```
.container {
    display: grid;
    grid-template-columns: repeat(4, 1fr);
    gap: 15px;
    grid-auto-rows: minmax(100px, auto);
    grid-template-areas:
        "header header header header"
        "bigBox content content content"
        "bigBox content content content"
        "footer footer footer footer";
}
```

Line placement isn't the only way we can place content into our grid.

- The **grid-template-areas** property allows us to give areas of our page a name.
- You lay this out using a string per row, with an area name for each column.
- As we have 4 columns, every string should contain 4 names – otherwise it won't work!

grid-template-areas



grid-template-areas

```
header {
    border-radius: 5px;
    padding: 10px;
    background-color: rgb(221, 225, 225);
    border: 2px solid rgb(58, 103, 103);
    text-align: center;
    grid-area: header;
}
```

- We then need to add a new property to each of our element selectors – **grid-area** – which will allow us to assign the element with one of the area names in our grid.
- Take a look at the **grid3** file in VSCode in comparison to **grid2.**
- We've made a few extra changes to our CSS/HTML to ensure **grid-template-areas** works as intended. What's changed?

Nesting Grids

The biggest change between **grid2** and **grid3** is that we've created a new <section> to contain our articles – and turned it into a **subgrid**.

```
.container {
        display: grid;
        grid-template-columns: repeat(4, 1fr);
        gap: 15px;
        grid-auto-rows: minmax(100px, auto);
        grid-template-areas:
                "header header header header"
                "bigBox content content content"
                "bigBox content content content"
                "footer footer footer footer";
}
```

```
.articleGrid {
        grid-area: content;
        display: grid;
        grid-template-columns: subgrid;
        grid-auto-rows:
        minmax(100px,150px);
        gap: inherit;
}
```

Nesting Grids

This is the area of our first grid – covering the entire container using the .container class.

We created our second grid inside the new <section> containing our articles. This is placed in the area named "content" within our first grid.

What are Media Queries?

The CSS **Media Query** allows us to create special rules for which browsers and devices our CSS gets applied to.

For example, you could apply rules based on screen size, or whether the displaying device has a touchscreen.

Media Query Syntax

Here's an example media query. Let's break it down:

```
@media screen and (width: 600px) {
    body {
        background-color: #ffffff;
    }
}
```

We start our query with the **@media at-rule**. We've seen at-rules before, so refer back to your Intro to CSS notes for a reminder!

Media Query Syntax

Media Queries are made up of three main parts:

```
@media screen and (width: 600px) {
    body {
        background-color: #ffffff;
    }
}
```

Media Type: Tells the browser whether this code is for screen media, print media, or both.

Media Expression: A condition that must be met before the CSS below can be applied.

Executed Code: The CSS rules that will be applied if the expression is **true**.

Media Types

There are three types of media you can specify in your media query:

print

Applies only to media when it is printed.

screen

Applies only to media when it is loaded in a browser on-screen

all

Applies to both printed and screen media. If you do not specify a type, this is the default.

Media Expressions

```
@media screen and (height: 1080px;) {}

@media screen and (max-width: 1920px) {}

@media (orientation: landscape) {}

@media (hover: hover) {}

@media (50% <= width <= 100%) {}
```

- Once we've targeted a media type, we can set our conditional expression.
- There are a range of different features we can use in our expression, from min or max width/height to device orientation.
- On the left are all examples of expressions you'll commonly see in media queries.

Media Expressions: Complex Logic

```
@media screen and (max-width: 1920px) and (orientation: landscape) {}

@media screen and (min-width: 1200px), screen and (orientation: landscape) {}

@media not (orientation: landscape) {}
```

- We can also combine expressions to create more advanced logic for our media query.
- Here we have examples of how we would incorporate **"and"**, **"or"**, and **"not" logic** into our expressions.

Breakpoints

```
@media screen and (max-width: 1920px) and (orientation: landscape) {}

@media screen and (min-width: 1200px), screen and (orientation: landscape) {}

@media not (orientation: landscape) {}
```

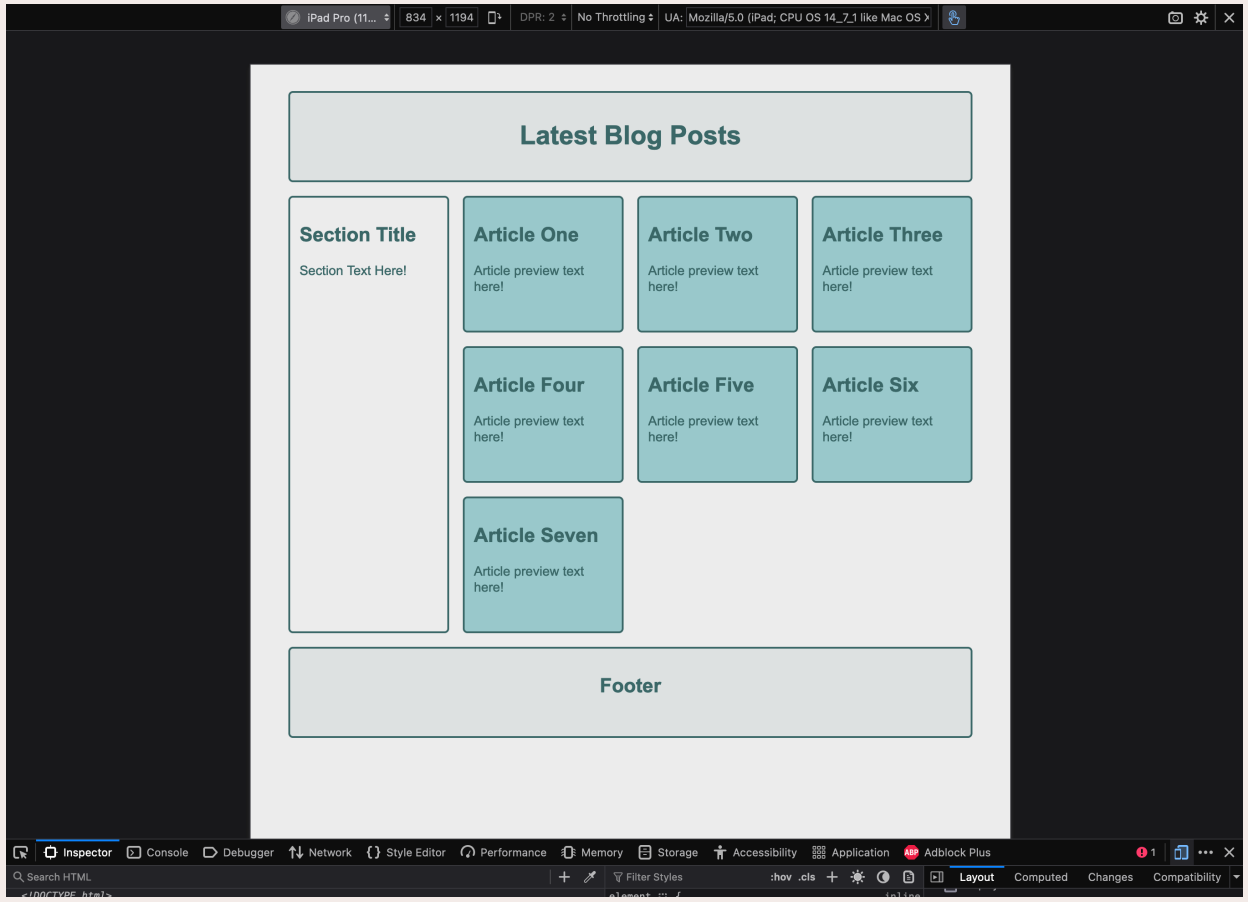
Media Queries were great in the early days of RWD, when there were only a handful of screen sizes designers needed to style for.

Nowadays, it's not realistic to be able to target *every* screen for *every* type of device.

Instead, it's more common to use media queries to target **breakpoints** – the screen sizes at which your content starts to break or look weird.

This way, we can cater for all size ranges without having to make a huge number of layouts.

Breakpoints

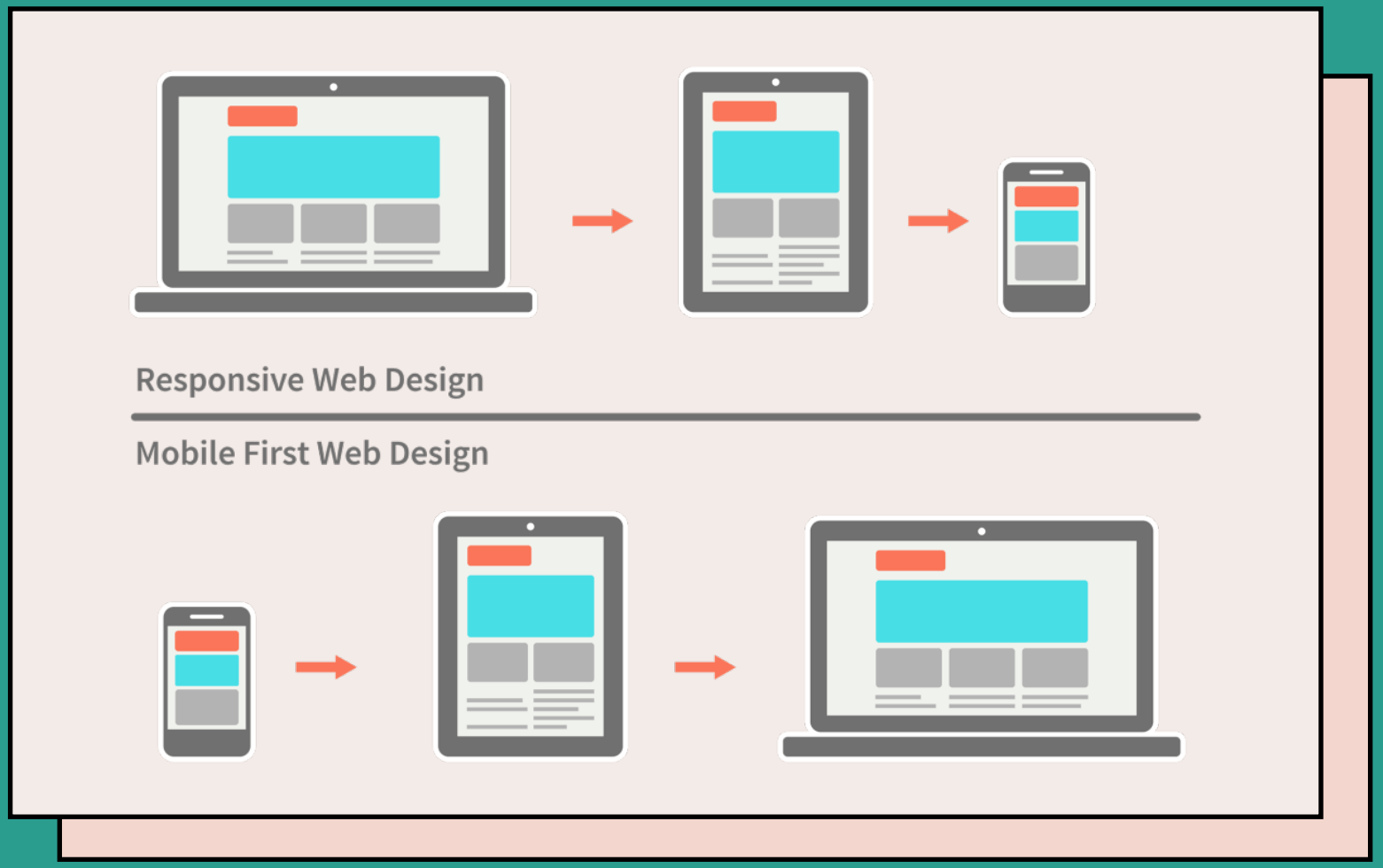


The **Developer Tools** of many web browsers will have a **Responsive Design** Mode, which will let you preview what your website looks like on different devices and screen sizes.

This is a great way to test out where your breakpoints should be.

Here, We're previewing our site on an 11-inch iPad Pro. If we switch to an iPhone, our site will probably look squashed or uneven, so we should add a media query to adjust it.

Mobile-first design



As you can see, there's a lot to think about when designing the layout of your page. To make it easier, there are two main approaches you can follow:

- Designing for your largest screen size, then making adjustments as viewport size reduces
- Designing for your smallest screen (often mobiles), then adding layout features as screen size increases

This second option is called **mobile-first design,** and is often the easier choice.

Additional Resources

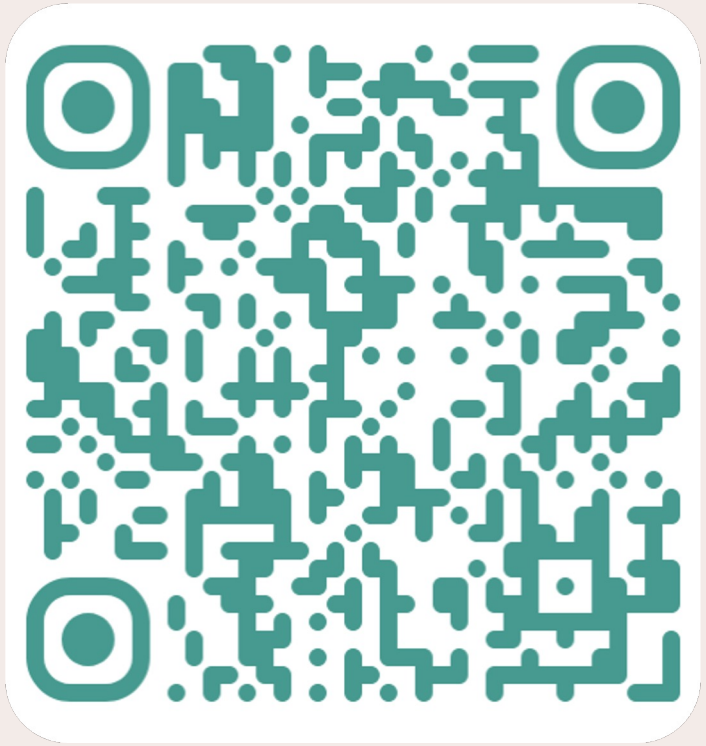
CSS Flexbox Tutorial Playlist || Net Ninja on Youtube

CSS Grid Intro || W3Schools

RWD: The Viewport || W3Schools

Learn Flexbox & Grid || CodeCademy

Do you really need a media query? || MDN Web Docs



Click/Scan the QR code to access the CTL VLE page!