Let's breakdown the `content` value of this responsive <meta> tag:

Values are comma separated, letting you specify a list of values for `content`

The `width` value is set to `device-width`. This will cause the browser to render the page at the same width of the device's screen size.

`initial-scale` set to `1` indicates the "zoom" value if your web page when it is first loaded. `1` means "no zoom."

There are other values you can specify for the `content` list -

```
<meta name="viewport" content="width=device-width, initial-scale=1.0”>
```

vh and vw

You can define height and width in terms of the viewport

- Use units **vh** and **vw** to set height and width to the percentage of the viewport's height and width, respectively

- 1vh = 1/100th of the viewport height

- 1vw = 1/100th of the viewport width

Example:

- height: 100vh; - width: 100vw;

Media Queries

the @media rule tells the browser to include a block of CSS properties only if a certain condition is true.

So this:

```
@media only screen and (max-width: 500px)  {
    body {
        background-color: light blue;
    }
}
```

Translates to:

```
if (the maximum width of the web page is 500 pixels) {
        then do this stuff
}
```

Media Queries

Breakpoint

```
/* For mobile phones: */
[class*="col-"] {
    width: 100%;
}
@media only screen and (min-width: 768px) {
    /* For desktop: */
    .col-1 {width: 8.33%;}
    .col-2 {width: 16.66%;}
    .col-3 {width: 25%;}
    .col-4 {width: 33.33%;}
    .col-5 {width: 41.66%;}
    .col-6 {width: 50%;}
    .col-7 {width: 58.33%;}
    .col-8 {width: 66.66%;}
    .col-9 {width: 75%;}
    .col-10 {width: 83.33%;}
    .col-11 {width: 91.66%;}
    .col-12 {width: 100%;
}
```

add a **breakpoint** where certain parts of the design will behave differently on each side of the breakpoint

many examples: https://www.w3schools.com/Css/css_rwd_mediaqueries.asp

rwd

Responsive Web Design

Most of these notes are verbatim txt from: Learning Web Design - Jennifer Niederst Robbins

Layout

Responsive design is based on fluid layouts that expand and contract to fill the available space in the viewport. One **fluid layout** is usually not enough, however, to serve all screen sizes. More often, two or three layouts are produced to meet requirements across devices, with small adjustments between layout shifts.

Designers typically start with a one-column layout that fits well on small handheld devices and rearrange elements into columns as more space is available. They may also have the design for the widest screens worked out early on so there is an end-point in mind. The design process may involve a certain amount of switching between views and making decisions about what happens along the way.

Responsive layout patterns

The manner in which a site transitions from a small-screen layout to a wide-screen layout must make sense for that particular site, but there are a few patterns (common and repeated approaches) that have emerged over the years. We can thank Luke Wroblewski (known for his “Mobile First” approach to web design, which has become the standard) for doing a survey of how responsive sites handle layout. Following are the top patterns Luke named in his **article**:

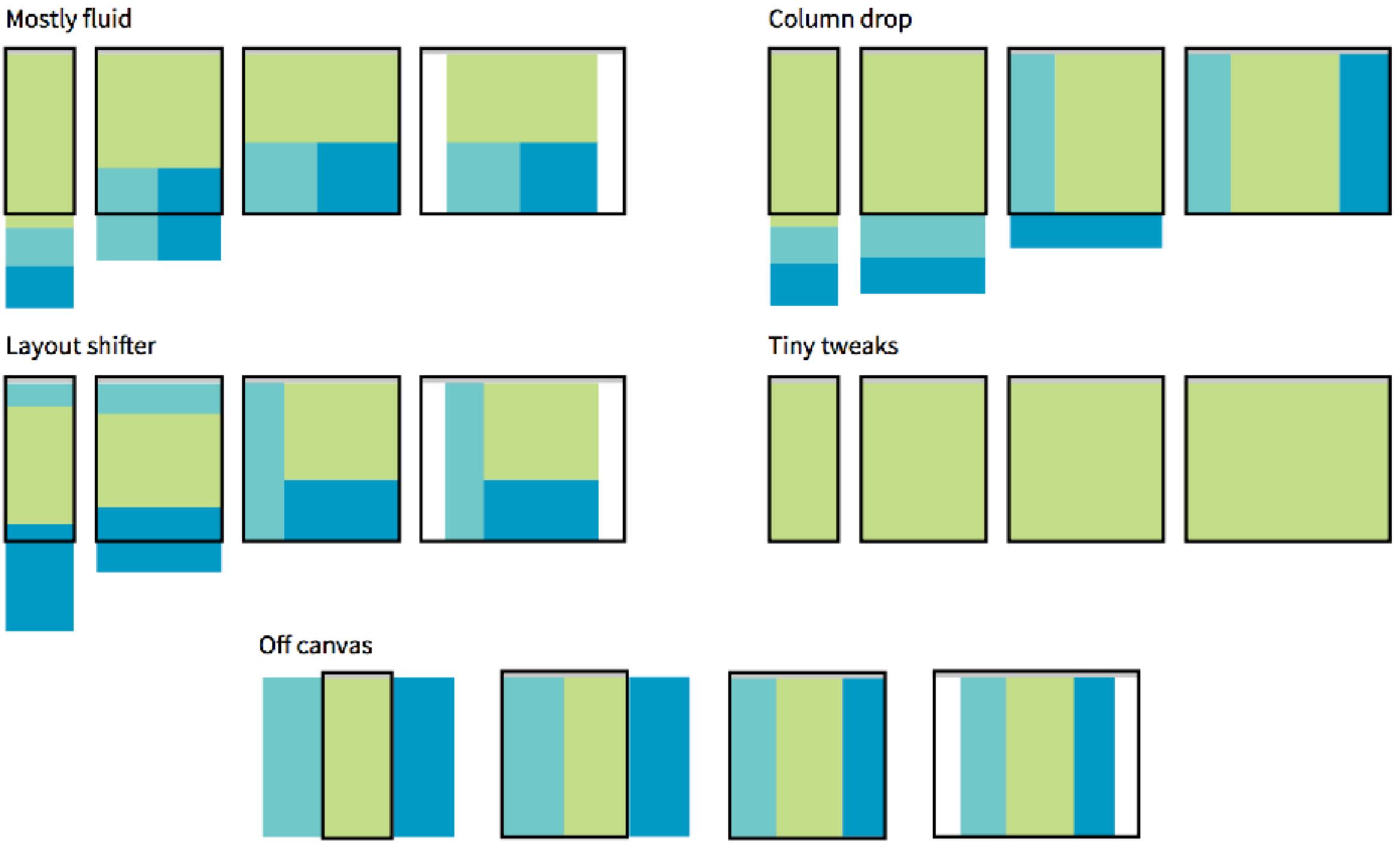


FIGURE 17-9. Examples of the responsive layout patterns identified by Luke Wroblewski.

Mostly fluid

This pattern uses a single-column layout for small screens, and another fluid layout that covers medium and large screens, with a maximum width set to prevent it from becoming too wide. It generally requires less work than other solutions.

Column drop

This solution shifts between one-, two-, and three-column layouts based on available space. When there isn't room for extra columns, the sidebar columns drop below the other columns until everything is stacked verti- cally in the one-column view.

Layout shifter

If you want to get really fancy, you can completely reinvent the layout for a variety of screen sizes. Although expressive and potentially cool, it is not necessary. In general, you can solve the problem of fitting your content to multiple environments without going overboard.

Tiny tweaks

Some sites use a single-column layout and make tweaks to type, spacing, and images to make it work across a range of device sizes.

Navigation

Navigation feels a little like the Holy Grail of Responsive Web Design. It is critical to get it right. Because navigation at desktop widths has pretty much been conquered, the real challenges come in re-creating our navigation options on small screens. A number of successful patterns have emerged for small screens, which I will briefly summarize here

Top navigation

If your site has just a few navigation links, they may fit just fine in one or two rows at the top of the screen.

Priority +

In this pattern, the most important navigation links appear in a line across the top of the screen alongside a More link that exposes additional options. The pros are that the primary links are in plain view, and the number of links shown can increase as the device width increases. The cons include the difficulty of determining which links are worthy of the prime small-screen real estate.

Display Property

The display property specifies if/how an element is displayed. Every HTML element has a default display value depending on what type of element it is. The default display value for most elements is block or inline. A block-level element always starts on a new line and takes up the full width available (stretches out to the left and right as far as it can). An inline element does not start on a new line and only takes up as much width as necessary.

```
display: none;
```

commonly used with JavaScript to hide/show elements without deleting and recreating them.

Overriding Default Display

Changing an inline element to a block element, or vice versa, can be useful for making the page look a specific way, and still follow the web standards.

```
li {
    display: inline;
}

span {
    display: block;
}
```

Note: Setting the display property of an element only changes how the element is displayed, NOT what kind of element it is. So, an inline element with display: block; is not allowed to have other block elements inside it.

Overflow Property

The CSS overflow property controls what happens to content that is too big to fit into an area. The overflow property specifies whether to clip content or to add scrollbars when the content of an element is too big to fit in a specified area. The overflow property only works for block elements with a specified height. The overflow property has the following values:

visible - Default. The overflow is not clipped. It renders outside the element's box
hidden - The overflow is clipped, and the rest of the content will be invisible
scroll - The overflow is clipped, but a scrollbar is added to see the rest of the content
auto - If overflow is clipped, a scrollbar should be added to see the rest of the content

Properties for left and right

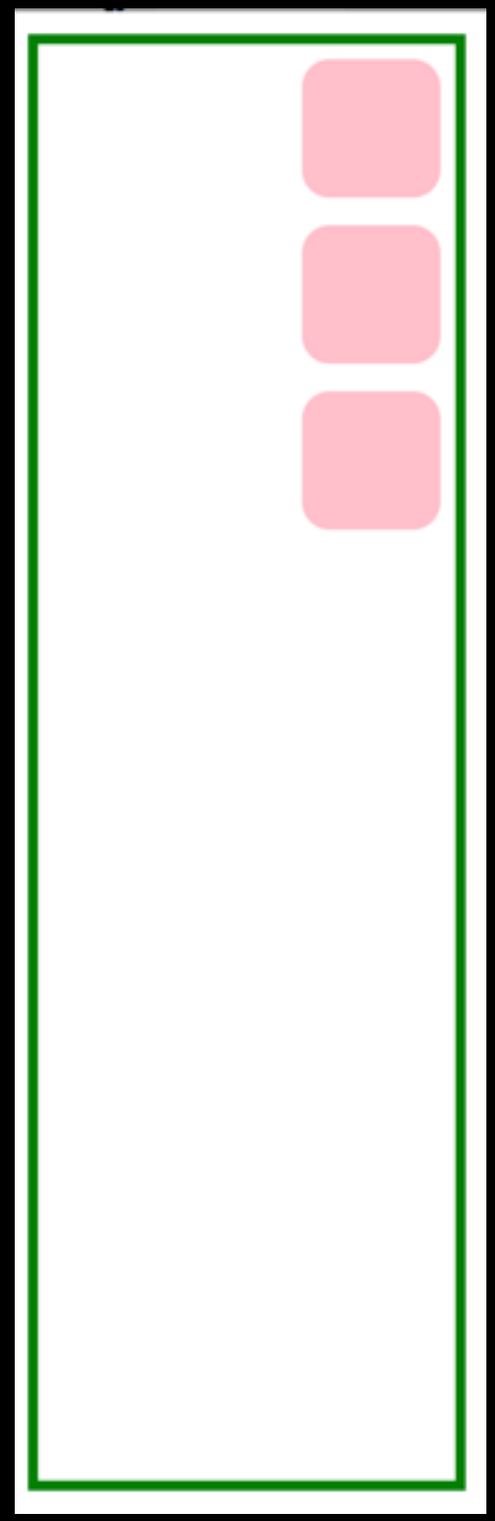
overflow-x
specifies what to do with the left/right edges of the content.

overflow-y
specifies what to do with the top/bottom edges of the content.

```
div.theExample1 {
    background-color: lightblue;
    height: 40px;
    width: 200px;
    overflow-y: scroll;
}

div.theExample2 {
    background-color: lightblue;
    height: 40px;
    width: 200px;
    overflow-y: hidden;
}
```

code from W3

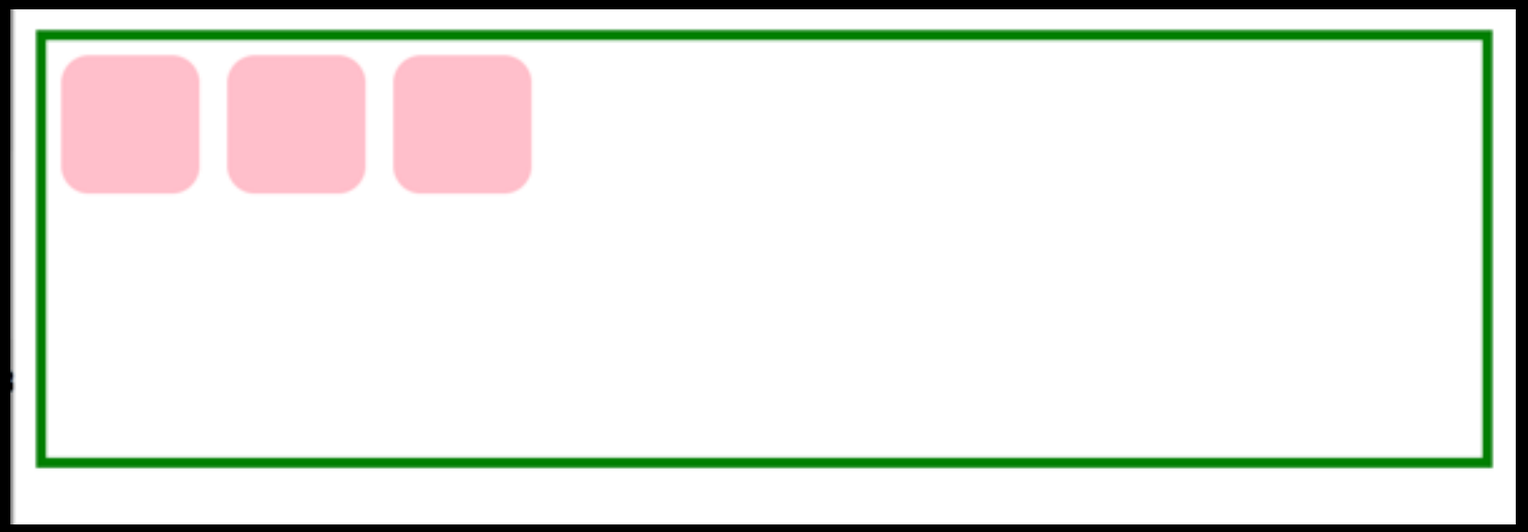
Flex Layout



To achieve more complicated layouts, we can enable a different kind of CSS layout rendering mode: **Flex layout.**

Flex layout defines a special set of rules for laying out items in rows or columns.

Flex Basics



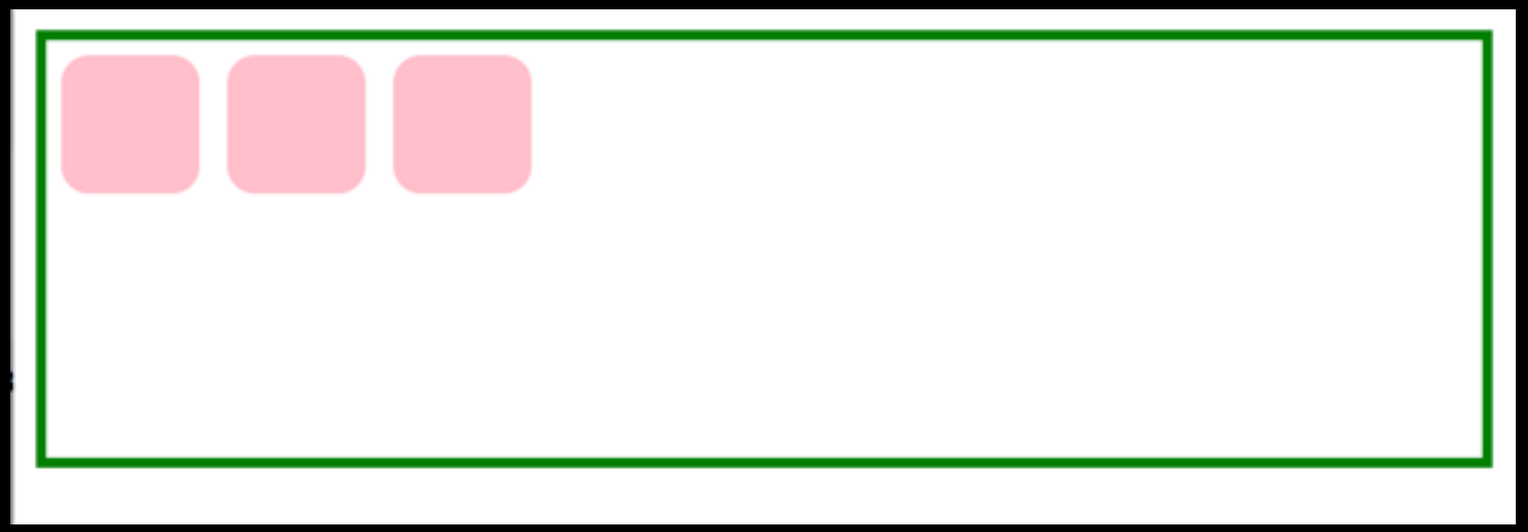
Flex layouts are composed of:

a **Flex container**, which contains one or more:

Flex item(s)

You can then apply CSS properties on the **Flex container** to dictate how the **Flex item(s)** are displayed

Flex Basics



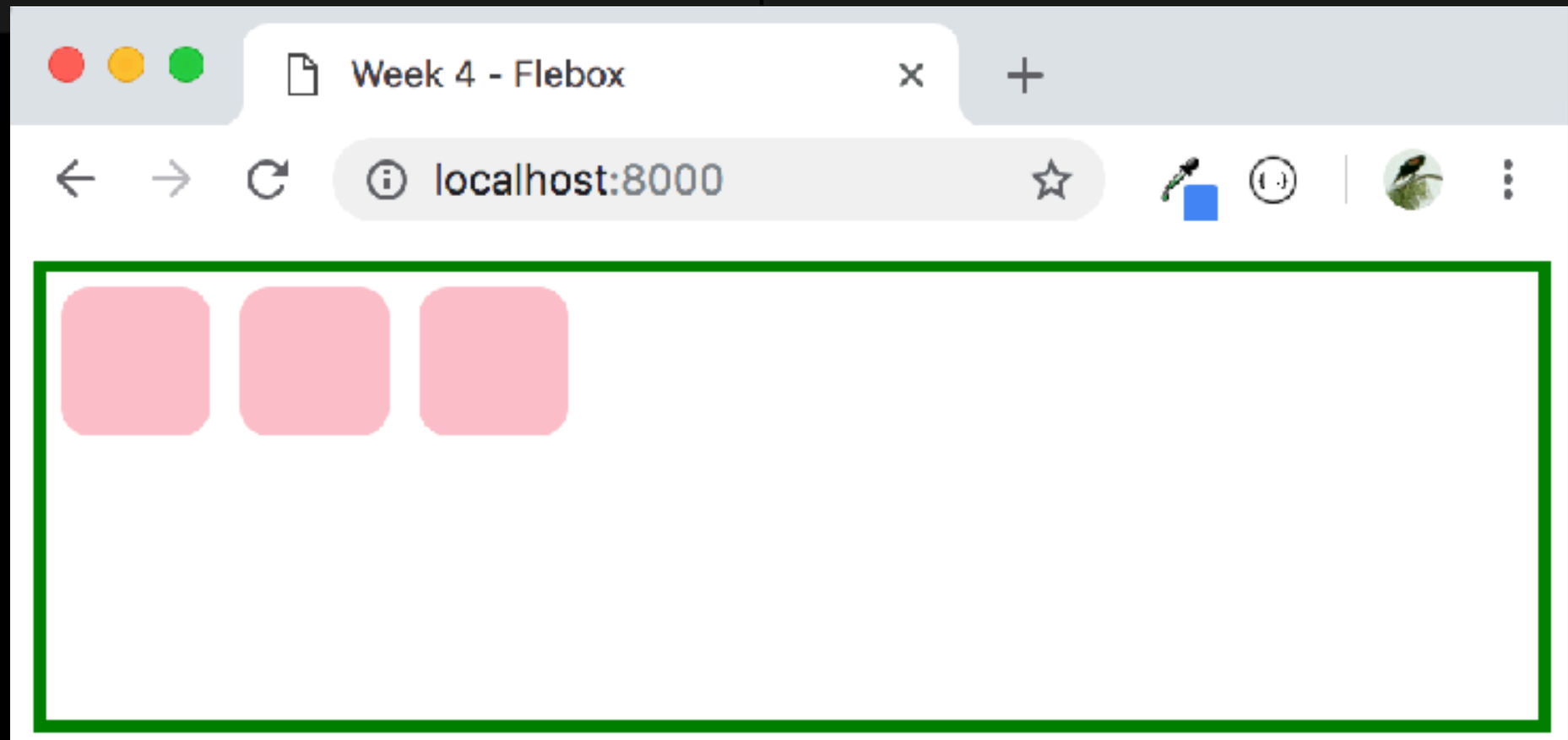
To make an element a flex container, change display:

- Block container: display: flex;
- Inline container: display: inline-flex;

Flex Basics

```
</head>

<body>

    <div id="flexBox">
     <div class="flexThing"></div>
     <div class="flexThing"></div>
     <div class="flexThing"></div>

    </div>
</body>
```

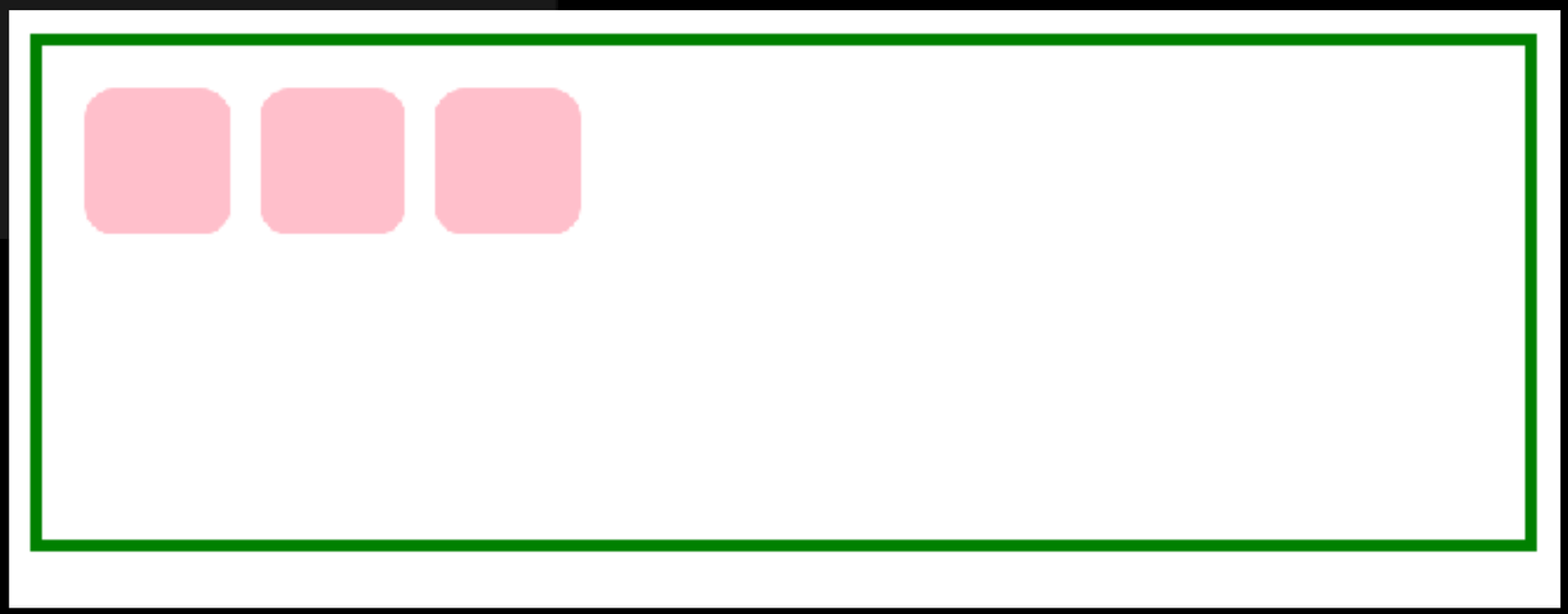
```
#flexBox {
    display: flex;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    border-radius: 10px;
    background-color: pink;
    height: 50px;
    width: 50px;
    margin: 5px;
}
```



Flex Basics: justify-content

You can control where the item is horizontally in the box by setting **justify-content** in the flex container.

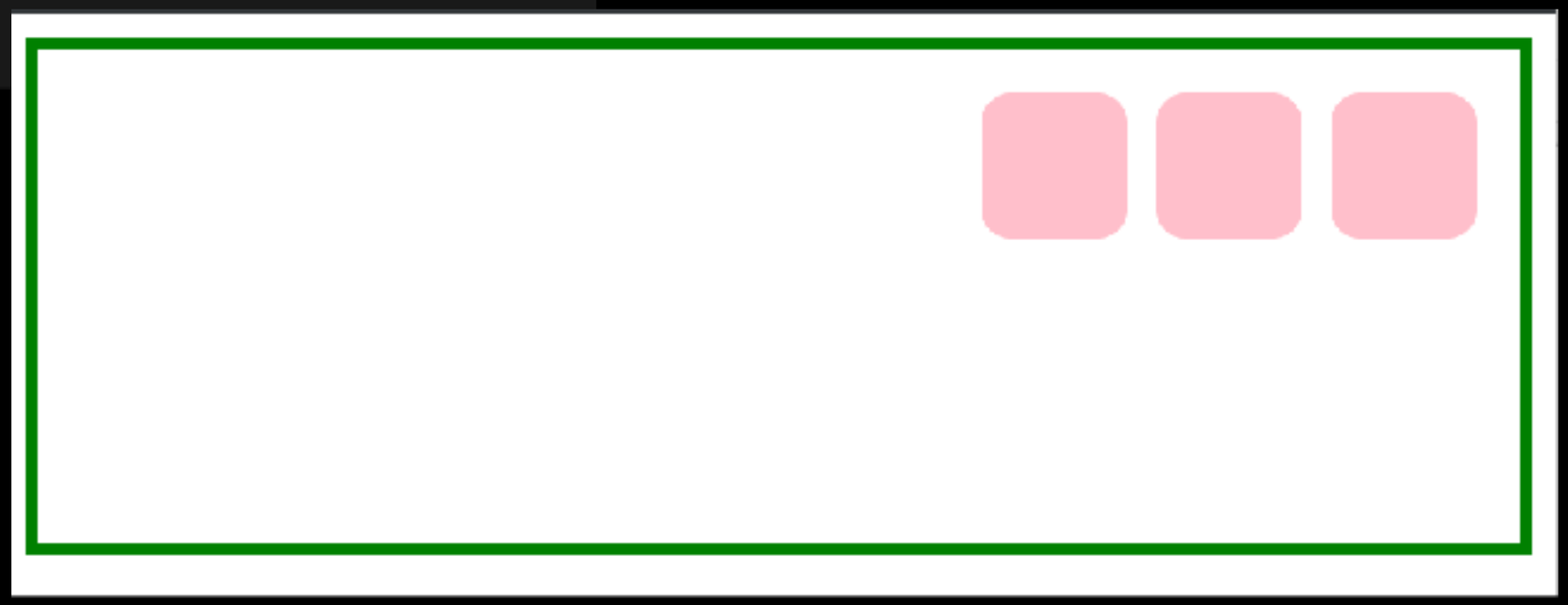
```
#flexBox {
    display: flex;
    border: 4px solid Green;
    justify-content: flex-start;
    padding: 10px;
    height: 150px;
}
```



Flex Basics: justify-content

You can control where the item is horizontally in the box by setting **justify-content** in the flex container.

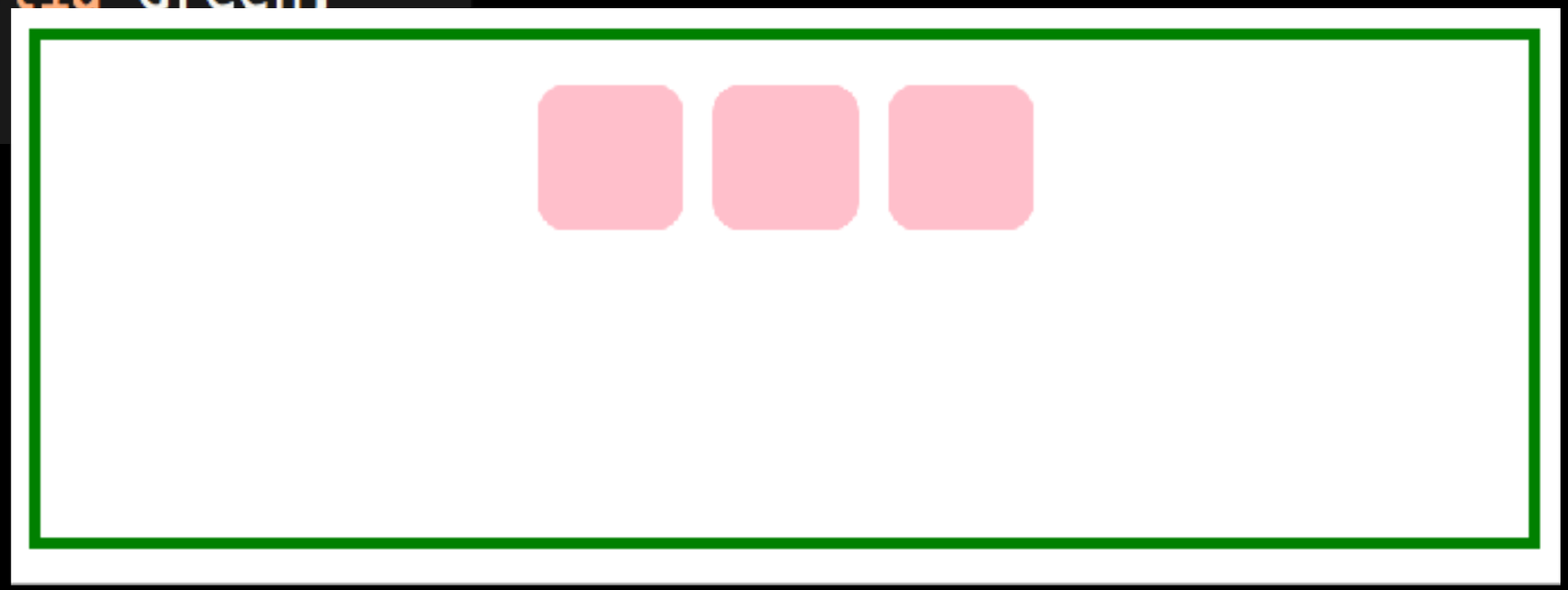
```
#flexBox {
    display: flex;
    justify-content: flex-end;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```



Flex Basics: justify-content

You can control where the item is horizontally in the box by setting **justify-content** in the flex container.

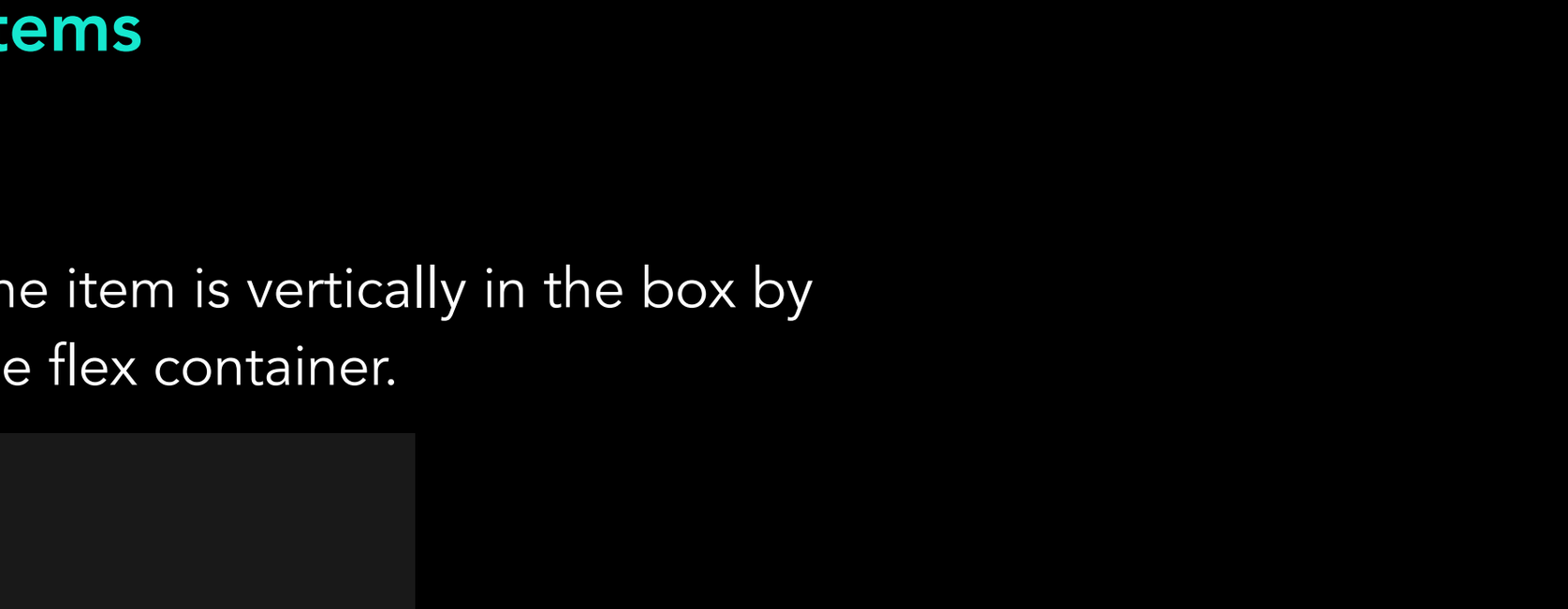
```
#flexBox {
    display: flex;
    justify-content: center;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```



Flex Basics: align-items

You can control where the item is vertically in the box by setting **align-items** in the flex container.

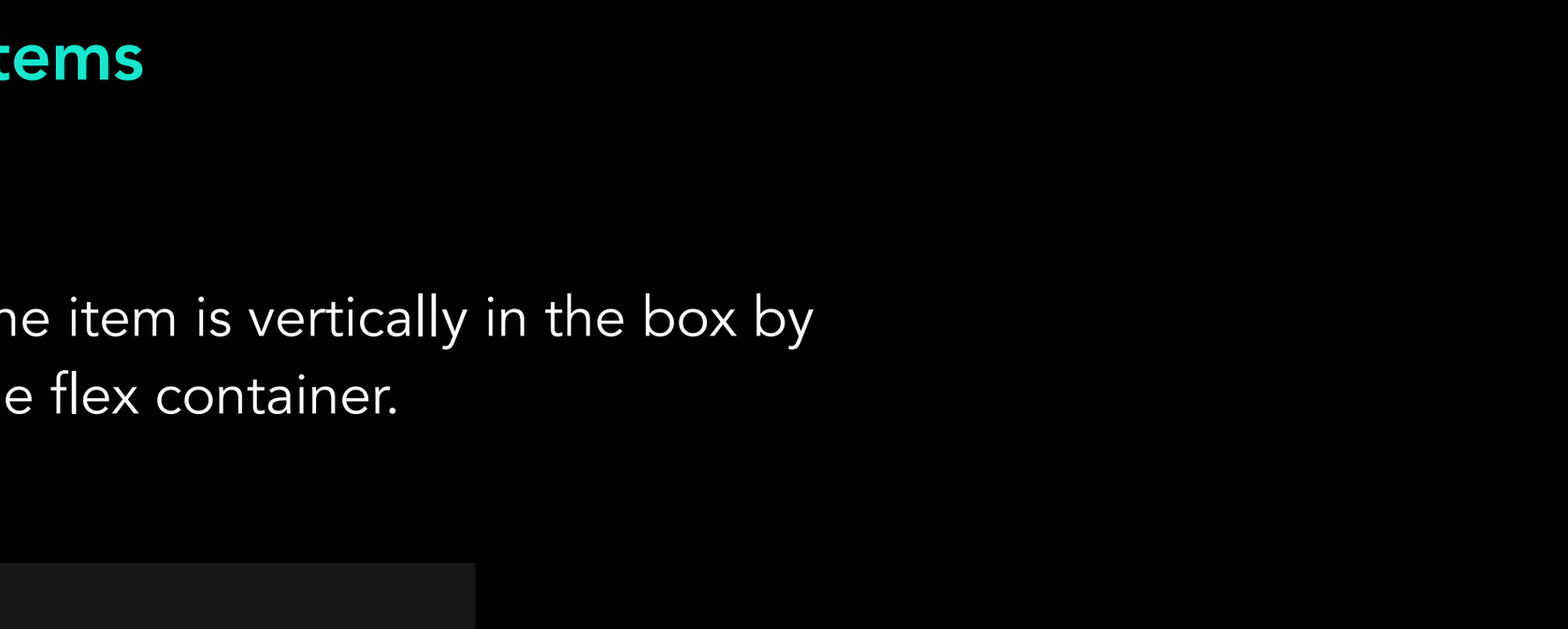
```
#flexBox {
    display: flex;
    align-items: flex-start;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```



Flex Basics: align-items

You can control where the item is vertically in the box by setting **align-items** in the flex container.

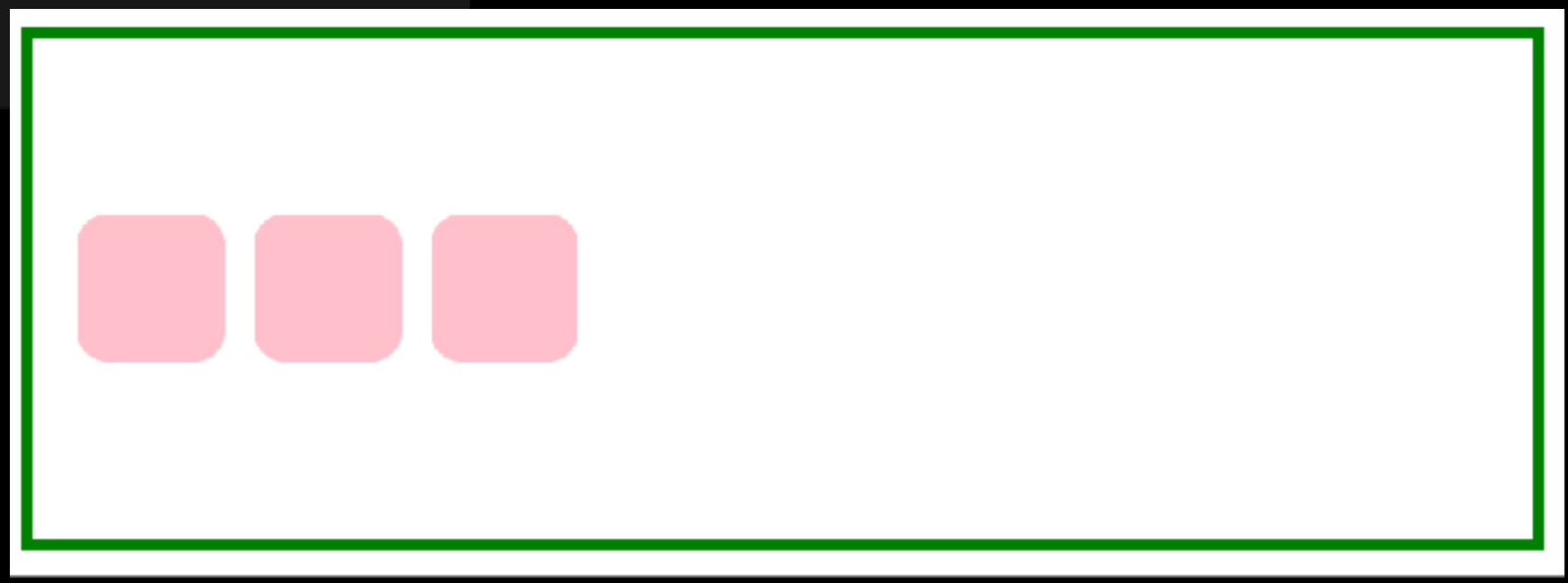
```
#flexBox {
    display: flex;
    align-items: flex-end;
    padding: 10px;
    height: 150px;
    border: 4px solid
}
```



Flex Basics: align-items

You can control where the item is vertically in the box by setting **align-items** in the flex container.

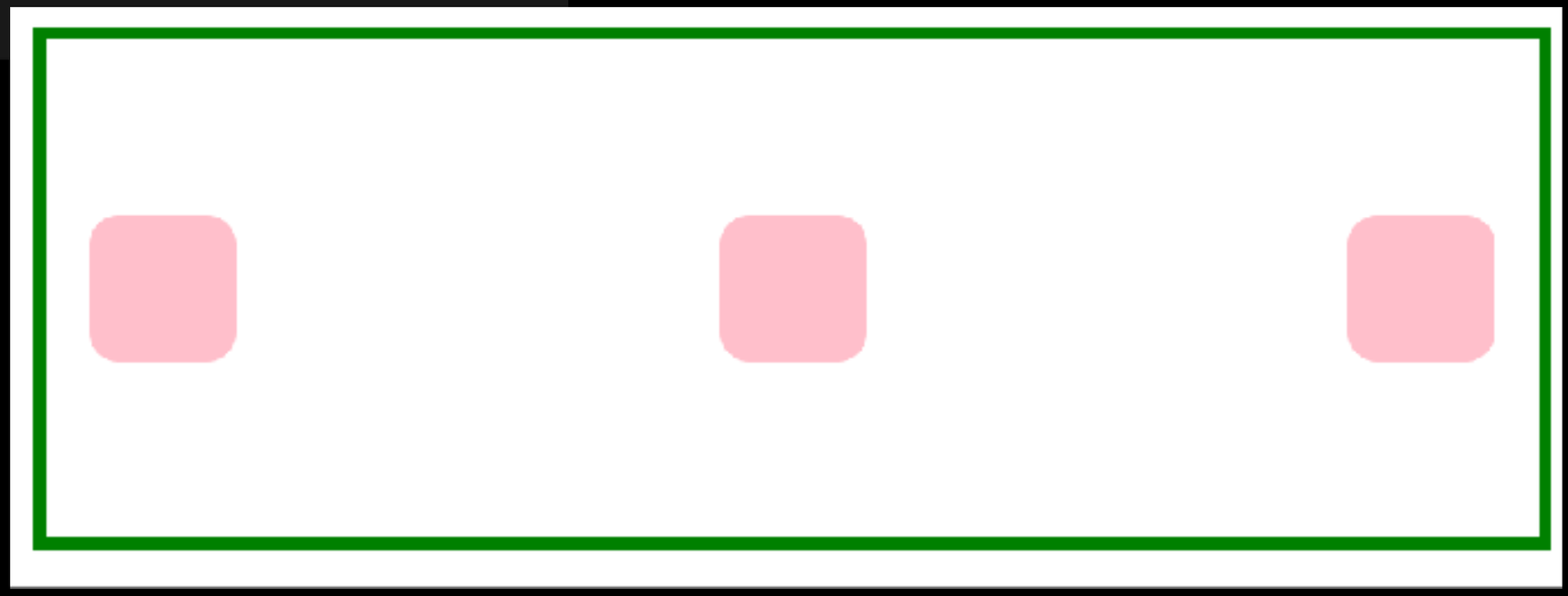
```
#flexBox {
    display: flex;
    align-items: center;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```



Flex Basics:

space-between + space-around

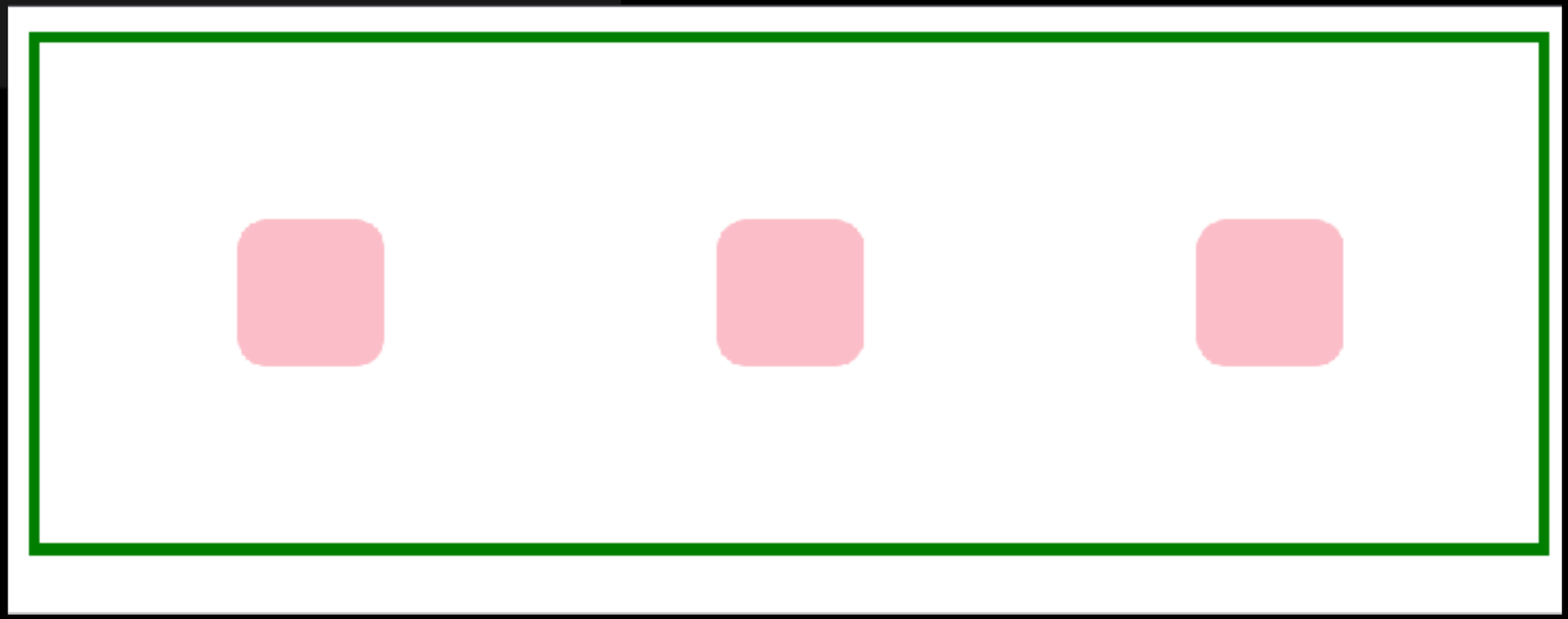
```
#flexBox {
    display: flex;
    justify-content: space-between;
    align-items: center;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```



Flex Basics:

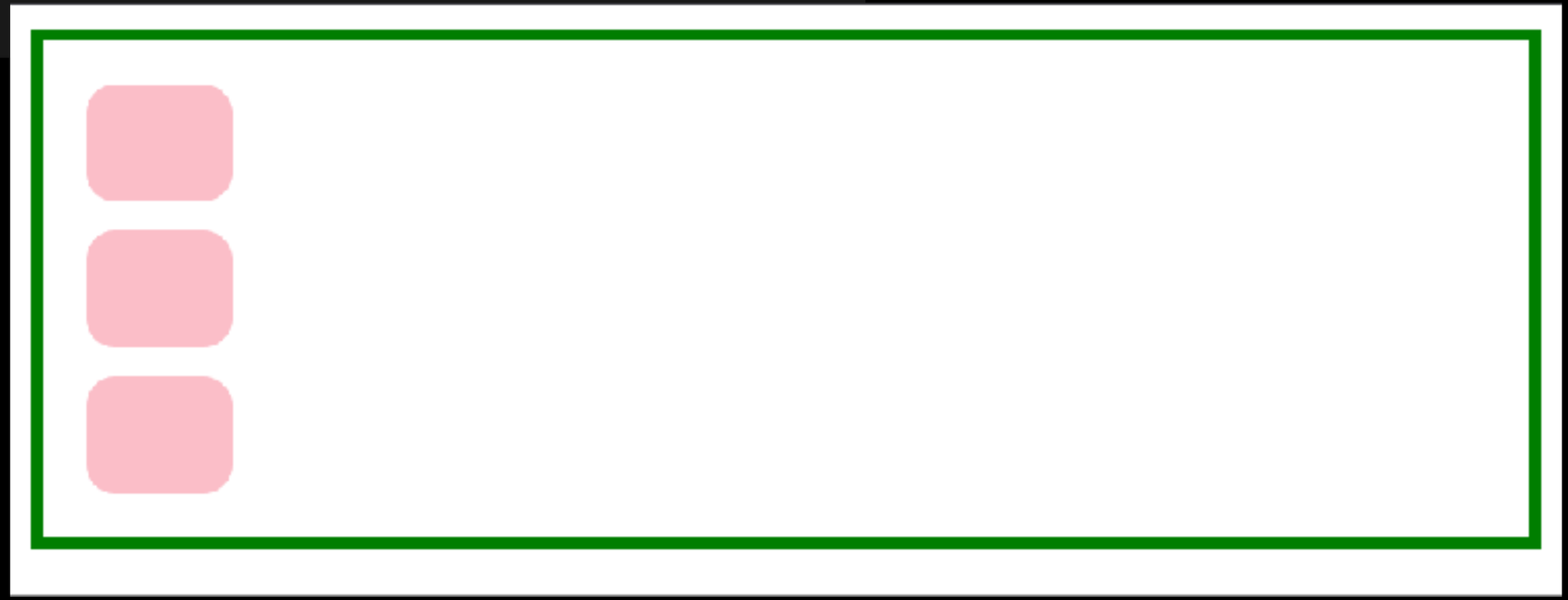
space-between + space-around

```
#flexBox {
    display: flex;
    justify-content: space-around;
    align-items: center;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```



Flex Basics: flex-direction

```
#flexBox {
    display: flex;
    flex-direction: column;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```



Flex Basics: flex-direction

```
#flexBox {
    display: flex;
    flex-direction: column;
    justify-content: center;
    padding: 10px;
    height: 300px;
    border: 4px solid Green;
}
```

Now **justify-content** controls where the column is vertically in the box.

Flex Basics: flex-direction

```
#flexBox {
    display: flex;
    flex-direction: column;
    justify-content: space-around;
    padding: 10px;
    height: 300px;
    border: 4px solid Green;
}
```

And you can also lay out columns instead of rows.

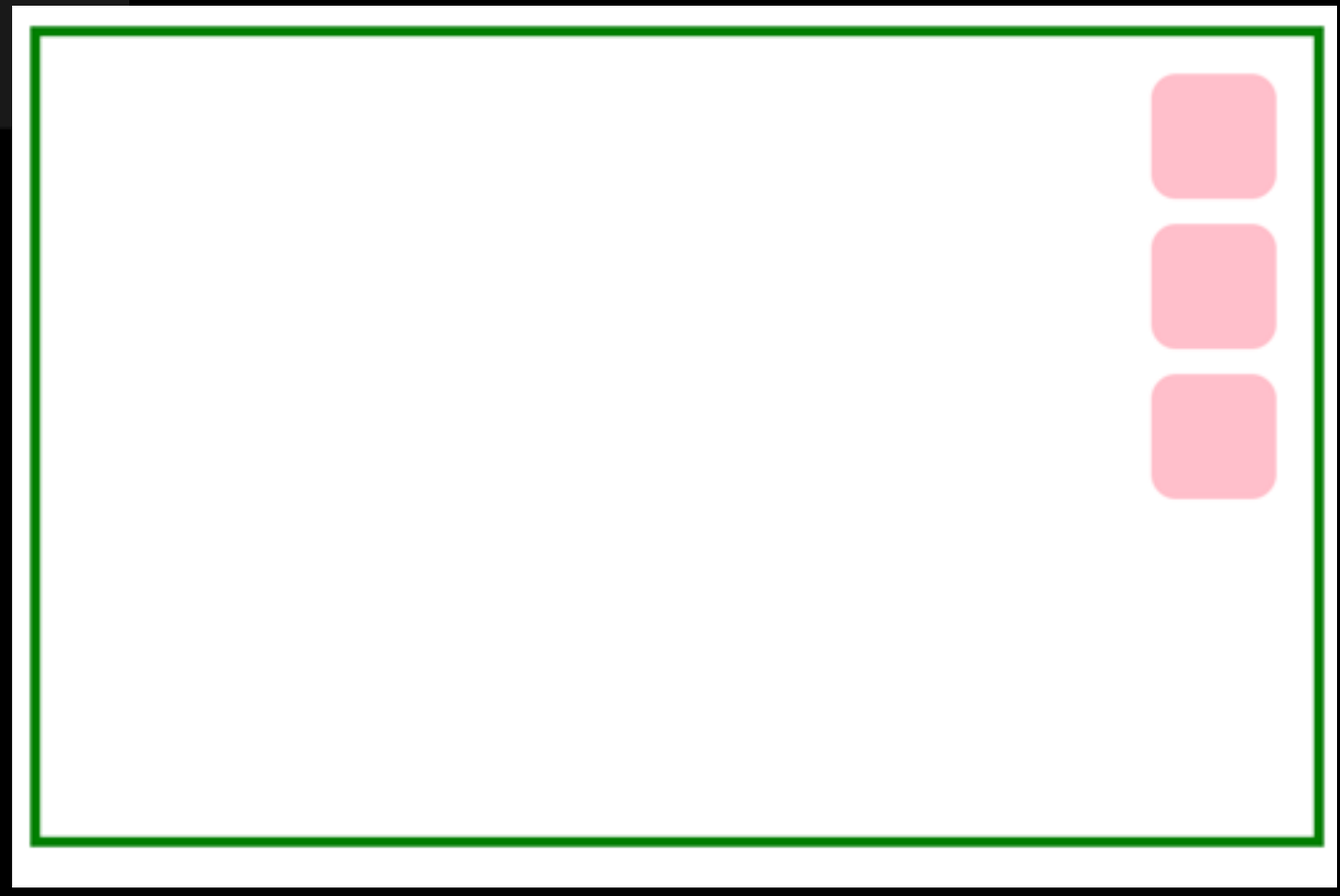
Now **justify-content** controls where the column is vertically in the box.

Flex Basics: flex-direction

```
#flexBox {
    display: flex;
    flex-direction: column;
    align-items: flex-end;
    padding: 10px;
    height: 300px;
    border: 4px solid Green;
}
```

And you can also lay out columns instead of rows.

Now **align-items** controls where the column is horizontally in the box.



Flex - different rendering model

When you set a container to **display: flex**, the direct children in that container are **flex items** + follow a new set of rules.

Flex items are not block or inline; they have different rules for their height, width + layout.

- The **contents** of a flex item follow the usual block/inline rules, relative to the flex item's boundary.

Flex Basis

Flex items have an initial width*, which, by default is either:

- The content width, or

- The explicitly set **width** property of the element, or

- The explicitly set **flex-basis** property of the element

This initial width* of the flex item is called the **flex basis**.

*width in the case of rows; height in the case of columns

Flex Basis

Flex items have an initial width*, which, by default is either:

- The content width, or

- The explicitly set **width** property of the element, or

- The explicitly set **flex-basis** property of the element

This initial width* of the flex item is called the **flex basis**.

The explicit width* of a flex item is respected **for all flex items**, regardless of whether the flex item is inline, block, or inline-block.

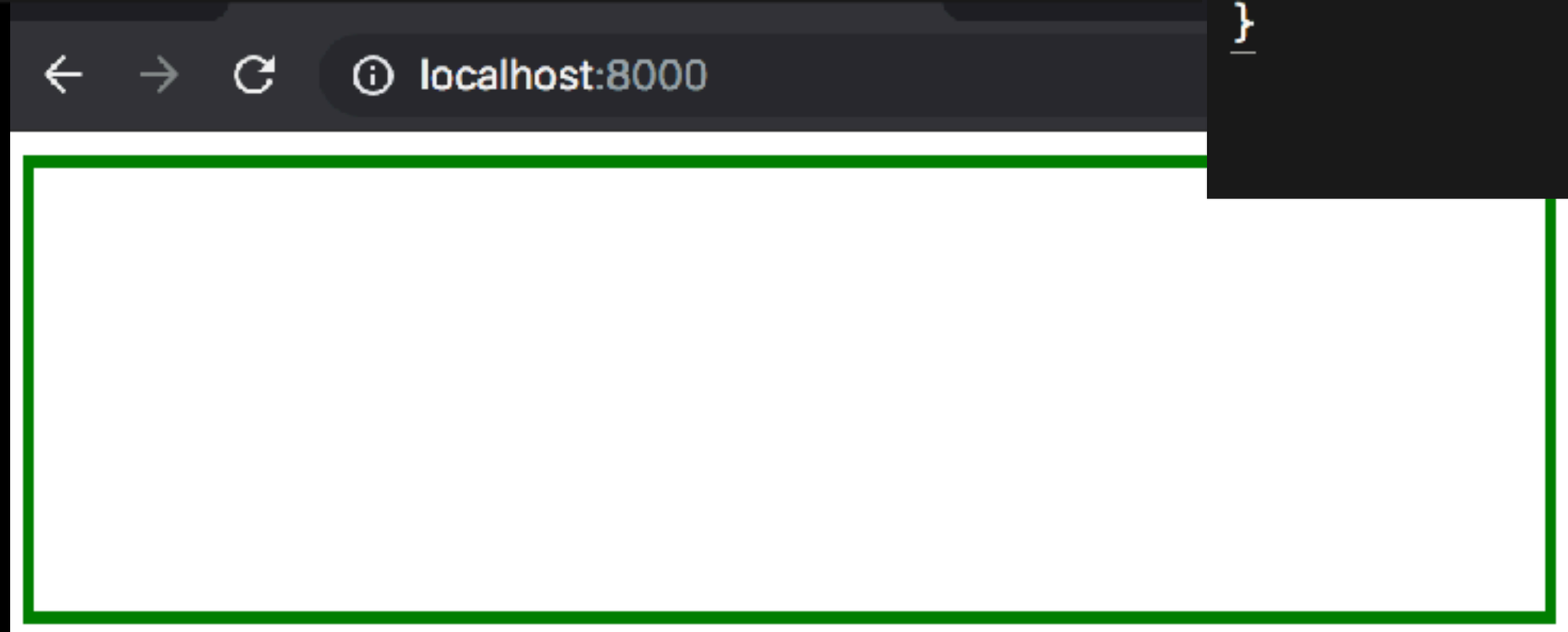
*width in the case of rows; height in the case of columns

Flex Basis

If we unset the height and width, our flex items disappears, because the **flex basis** is now the content size, which is empty:

```
<div id="flexBox">
 <span class="flexThing"></span>
 <div class="flexThing"></div>
 <span class="flexThing"></span>
</div>
```

```
#flexBox {
    display: flex;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    border-radius: 10px;
    background-color: pink;
    margin: 5px;
}
```



flex-shrink

The width* of the flex item can automatically shrink **smaller** than the **flex basis** via the **flex-shrink** property:

flex-shrink:

- If set to **1**, the flex item shrinks itself as small as it can in the space available
- If set to **0**, the flex item does not shrink.

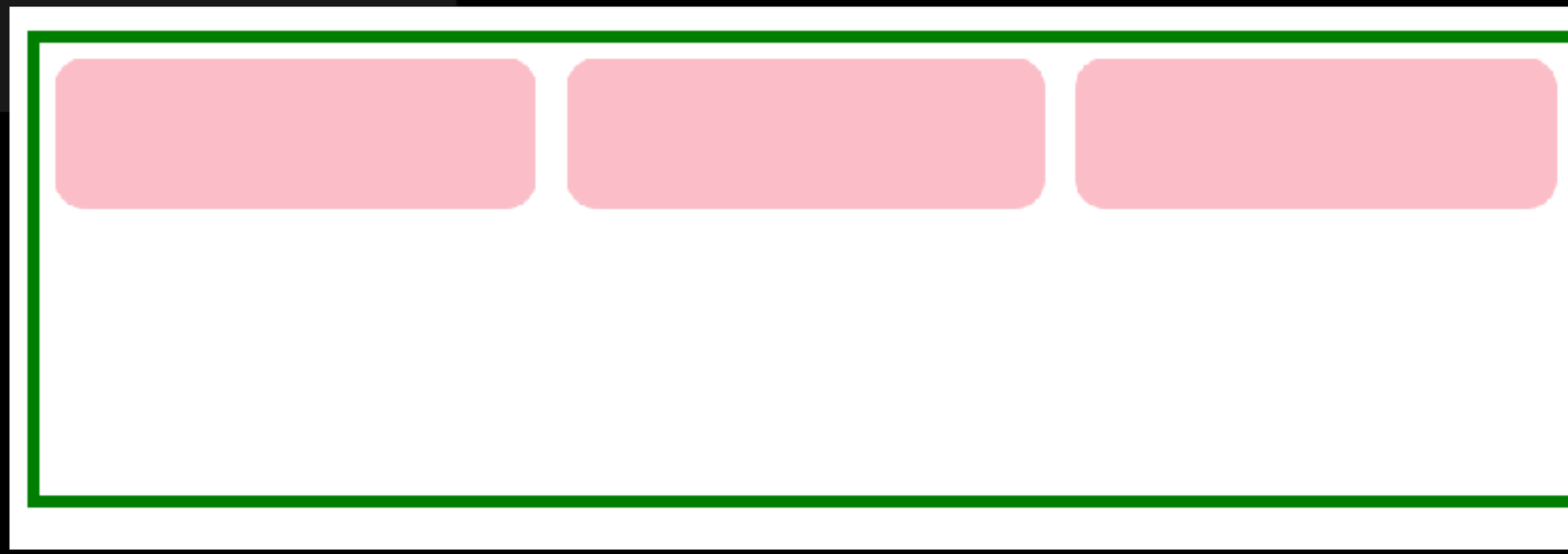
Flex items have **flex-shrink**: **1 by default**.

*width in the case of rows;
height in the case of columns

flex-shrink

```
#flexBox {
    display: flex;
    align-items: flex-start;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    width: 500px;
    height: 50px;
    border-radius: 10px;
    background-color: pink;
    margin: 5px;
}
```

The flex items' widths all shrink to fit the width of the container.



flex-shrink

```
.flexThing {
    width: 500px;
    height: 50px;
    flex-shrink: 0;
    border-radius: 10px;
    background-color: pink;
    margin: 5px;
}
```

Setting **flex-shrink**: **0**;
undoes the shrinking behavior,
and the flex items do not
shrink in any circumstance:

flex-grow

The width* of the flex item can automatically **grow larger** than the **flex basis** via the **flex-grow** property:

flex-grow:

- If set to **1**, the flex item grows itself as large as it can in the space remaining
- If set to **0**, the flex item does not grow

Flex items have **flex-grow**: **0 by default**.

*width in the case of rows;
height in the case of columns

flex-grow

Let’s unset the height + width of our flex items again.

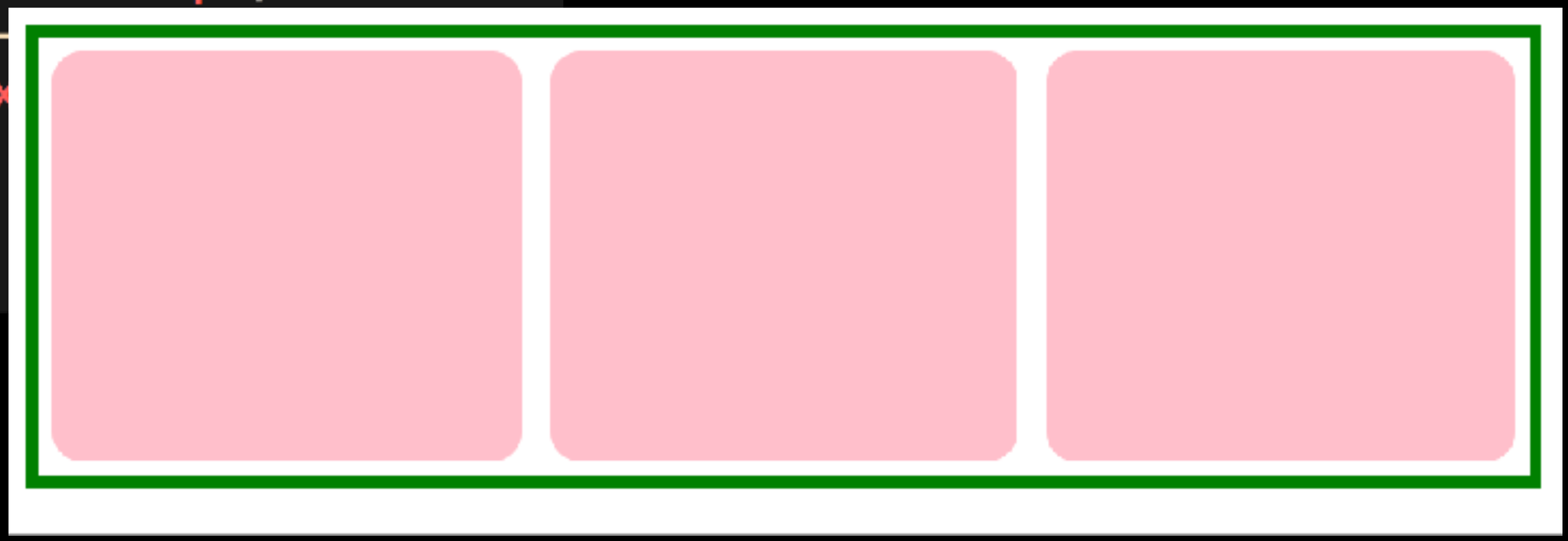
```
<div id="flexBox">
 <span class="flexThing"></span>
 <div class="flexThing"></div>
 <span class="flexThing"></span>
</div>
```

```
#flexBox {
    display: flex;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    border-radius: 10px;
    background-color: pink;
    margin: 5px;
}
```

flex-grow

if we set **flex-grow: 1;**
the flex items fill the empty space.

```
#flexBox {
    display: flex;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    flex-grow: 1;
    border-radius: 10px;
    background-
    margin: 5px
}
```

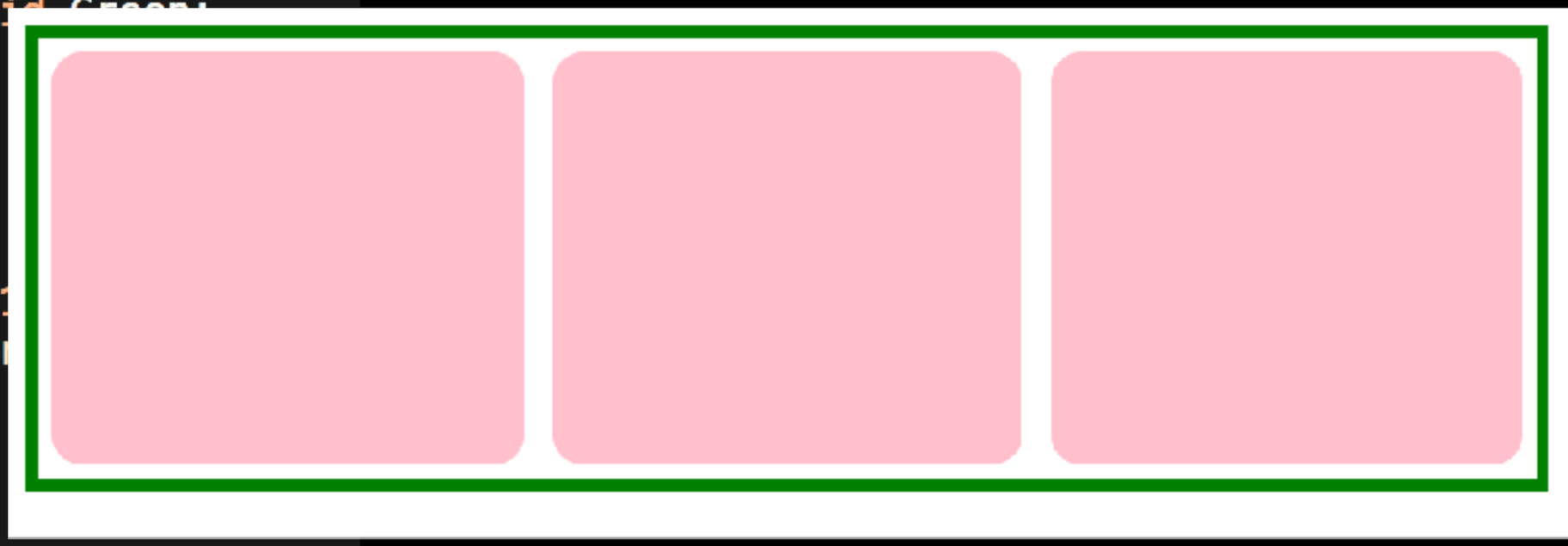


flex item height?**

note that **flex-grow** only controls width*

So why does the height** of the flex items seem to ‘grow’ as well?

```
#flexBox {
    display: flex;
    border: 4px soli
    height: 150px;
}

.flexThing {
    flex-grow: 1;
    border-radius:
    background-colo
    margin: 5px;
}
```



*width in the case of rows; height in the case of columns

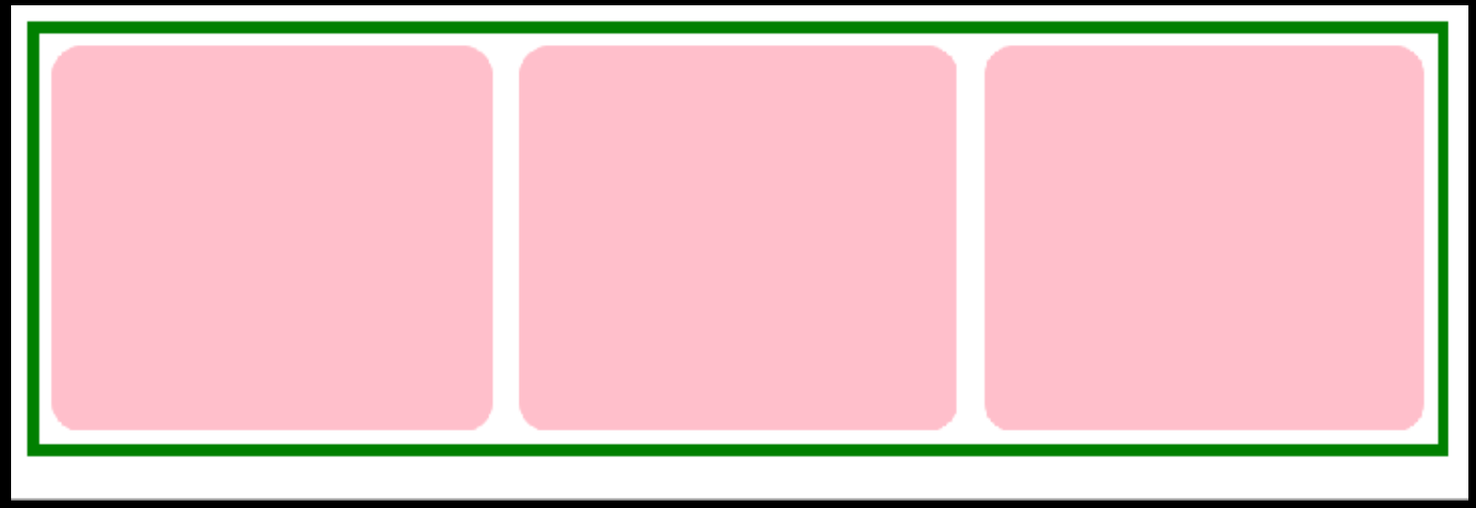
**height in the case of rows; width in the case of columns

align-items: stretch;

The default value of **align-items** is stretch, which means every flex item grows vertically* to fill the container by default.

(This will not happen if the height on the flex item is set)

```
#flexBox {
    display: flex;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    flex-grow: 1;
    border-radius: 10px;
    background-color: pink;
    margin: 5px;
}
```



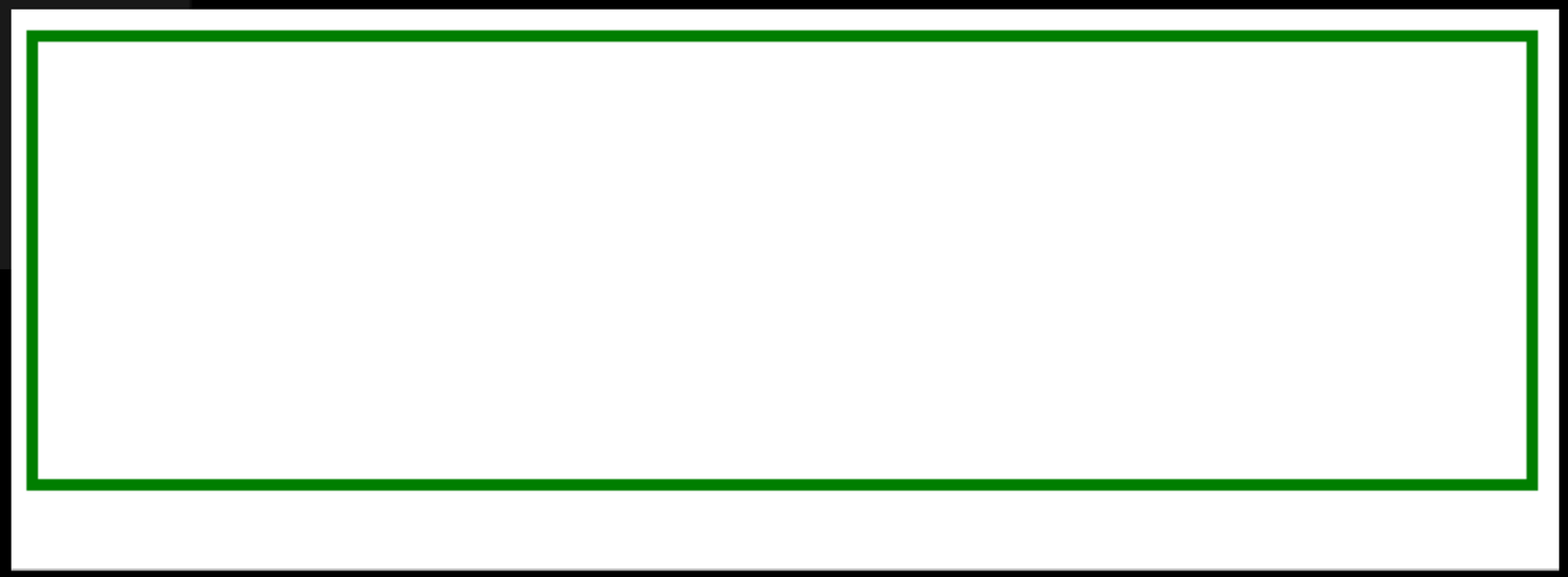
*veritcally in the case of rows; horizontally in the case of columns

align-items: stretch;

If we set another value for align-items, the flex items disappear again because the height is now content height, which is 0:

```
#flexBox {
    display: flex;
    align-items: flex-start;
    border: 4px solid Green;
    height: 150px;
}
```

```
.flexThing {
    flex-grow: 1;
    border-radius: 10px;
    background-color: pink;
    margin: 5px;
}
```



css - grids

Flexbox & CSS Grid

"The basic difference between CSS Grid Layout and CSS Flexbox Layout is that flexbox was designed for layout in one dimension - either a row or a column. Grid was designed for two-dimensional layout - rows, and columns at the same time. The two specifications share some common features, however, and if you have already learned how to use flexbox, the similarities should help you get to grips with Grid."

MDN

The flex-direction property defines in which direction the container wants to stack the flex items - either flex-direction: row or flex-direction: column. However, by using flex-wrap property. Read all about **CSS Flexbox @ W3**.

CSS Grid

A grid is an intersecting set of horizontal and vertical lines - one set defining columns and the other rows. Elements can be placed onto the grid, respecting these column and row lines.

How Grid Layout Works

The process for using the CSS Grid Layout Module is fundamentally simple:

+ Use the display property to turn an element into a grid container. The element's children automatically become grid items.
+ Set up the columns and rows for the grid. You can set them up explicitly and/or provide directions for how rows and columns should get created on the fly (the css grid is very flexible).
+ Assign each grid item to an area on the grid. If you don't assign them explicitly, they flow into the cells sequentially.

The element that has the display: **grid property** applied to it becomes the grid container and defines the context for grid formatting. All of its direct child elements automatically become grid items that end up positioned in the grid. You can define an explicit grid with grid layout but the specification also deals with the content added outside of a declared grid, which adds additional rows and columns when needed. Features such as adding "as many columns that will fit into a container" are included.

Grid line

The horizontal and vertical dividing lines of the grid are called grid lines.

Grid cell

The smallest unit of a grid is a grid cell, which is bordered by four adjacent grid lines with no grid lines running through it.

Grid area

A grid area is a rectangular area made up of one or more adjacent grid cells.

Grid track

The space between two adjacent grid lines is a grid track, which is a generic name for a grid column or a grid row. Grid columns are said to go along the block axis, which is vertical (as block elements are stacked) for languages written horizontally. Grid rows follow the inline (horizontal) axis.

The structure established for the grid is independent from the number of grid items in the container. You could place 4 grid items in a grid with 12 cells, leaving 8 of the cells as 'whitespace.' That's the flexibility of grids. You can also set up a grid with fewer cells than grid items, and the browser adds cells to the grid to accommodate them.

Grid Container Properties:

display

grid-template-columns

grid-template-rows

grid-template-areas

grid-template

grid-column-gap

grid-row-gap

grid-gap

justify-items

align-items

place-items

justify-content

align-content

place-content

grid-auto-columns

grid-auto-rows

grid-auto-flow

grid

CSS Tricks w/ links!

Grid Item Properites

grid-column-start

grid-column-end

grid-row-start

grid-row-end

grid-column

grid-row

grid-area

justify-self

align-self

place-self

CSS Grid Functions

repeat ( )

minmax ( )

fit-content ( )

Fr unit

flexible length

image files for the web

SVG

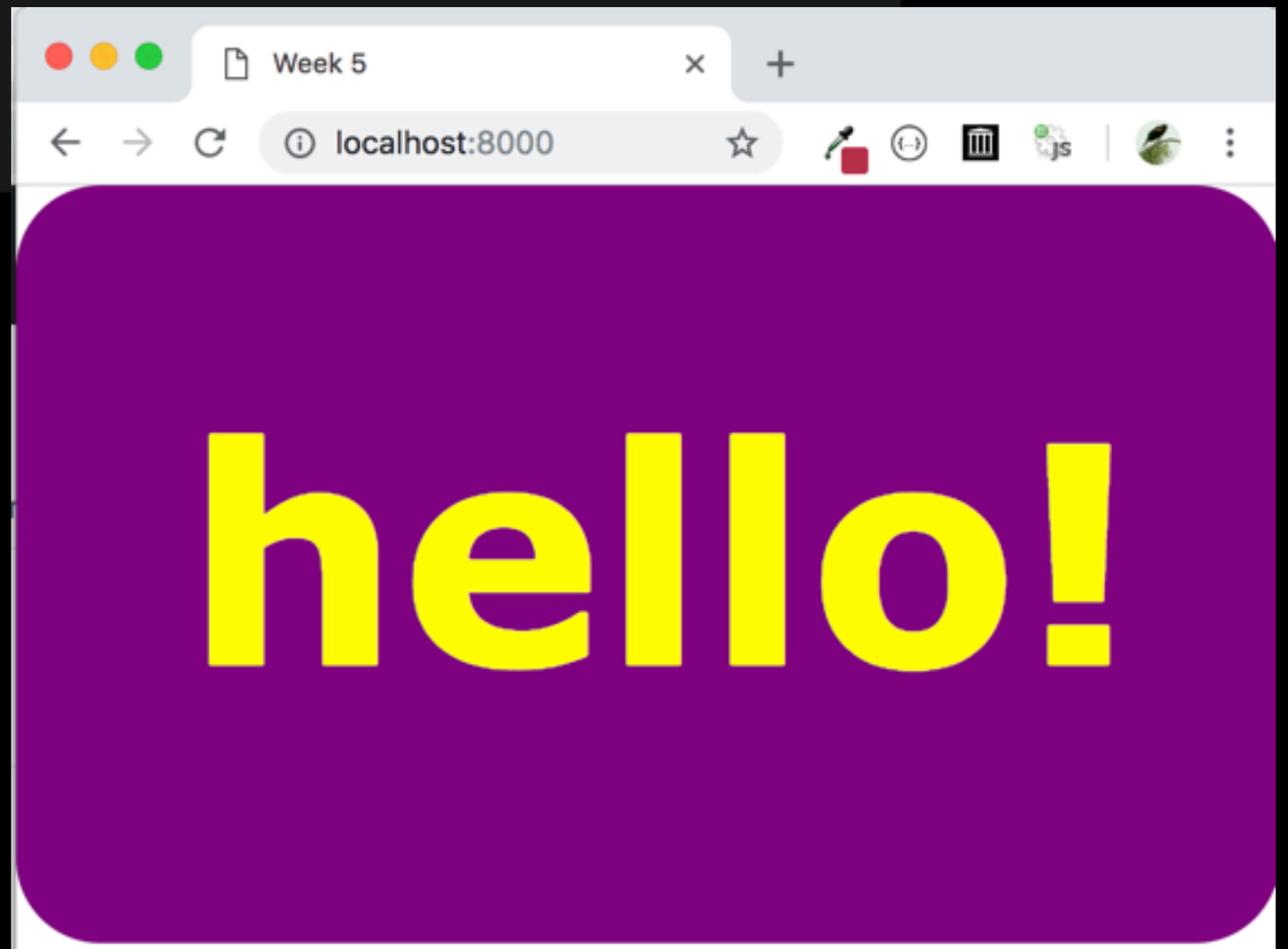
SVG is a 2D vector imaged based on an **XML** (Extensible Markup Language) syntax. It provides the rules and standards for how markup languages should be written and work together. As a result, SVG works well with HTML content.

In SVGs shapes and paths are specified by instructions written out in a text file. Let that sink in: they are images that are written out in text! All of the shapes and paths as well as their properties are written out in the standardized SVG markup language. As HTML has elements for paragraphs **<p>** and navigation **<table>**, SVG has elements that define shapes like rectangle **(rect)**, circle **(circle)**, and paths **(path)**.

A simple example will give you the general idea. Here is the SVG code that describes a rectangle **(rect)** with rounded corners (rx and ry, for x-radius and y-radius) and the word "hello" set as text with attributes for the font and color. Browsers that support SVG read the instructions and draw the image exactly as designed:

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 300 180">
  <rect width="300" height="180" fill="purple" rx="20" ry="20"/>
  <text x="40" y="114" fill="yellow" font-family="'Verdana-Bold'"
  font-size="72">
     hello!
  </text>
</svg>
```

```
rect:hover{
   fill:green;
}
```



This is before responsive web design.

SVG

Advantages of SVGs over bitmapped counterparts for certain image types:

Because they save only instructions for what to draw, they generally require **less data** than an image saved in a bitmapped format. That means faster downloads and better performance.

Because they are **vectors,** they can resize as needed in a responsive layout without loss of quality. An SVG is always nice and crisp. No fuzzy edges.

Because they are text, they integrate well with HTML/XML and can be compressed with tools like Gzip and Brotli, just like HTML files.

They can be animated.

You can change how they look with CSS.

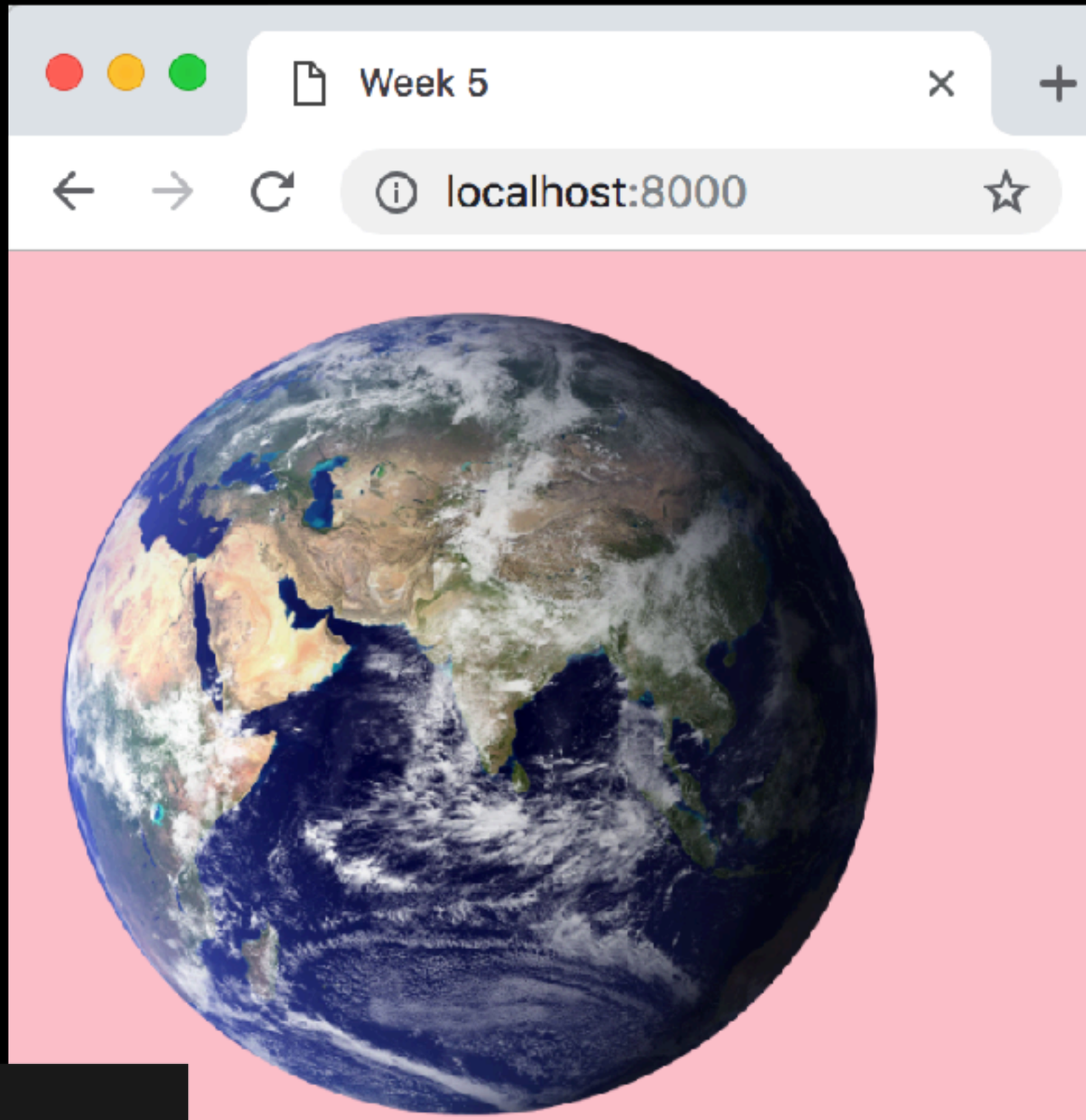
You can add interactivity with JavaScript so you can add interaction design.

PNG

A raster image is a grid of pixels. Each discreet pixels each have an (R,G,B,A)

red
green
blue
alpha - transparency,
0 -1
in this case it is set to 0.

Using the Canvas we can start to manipulate pixels using Javascript.



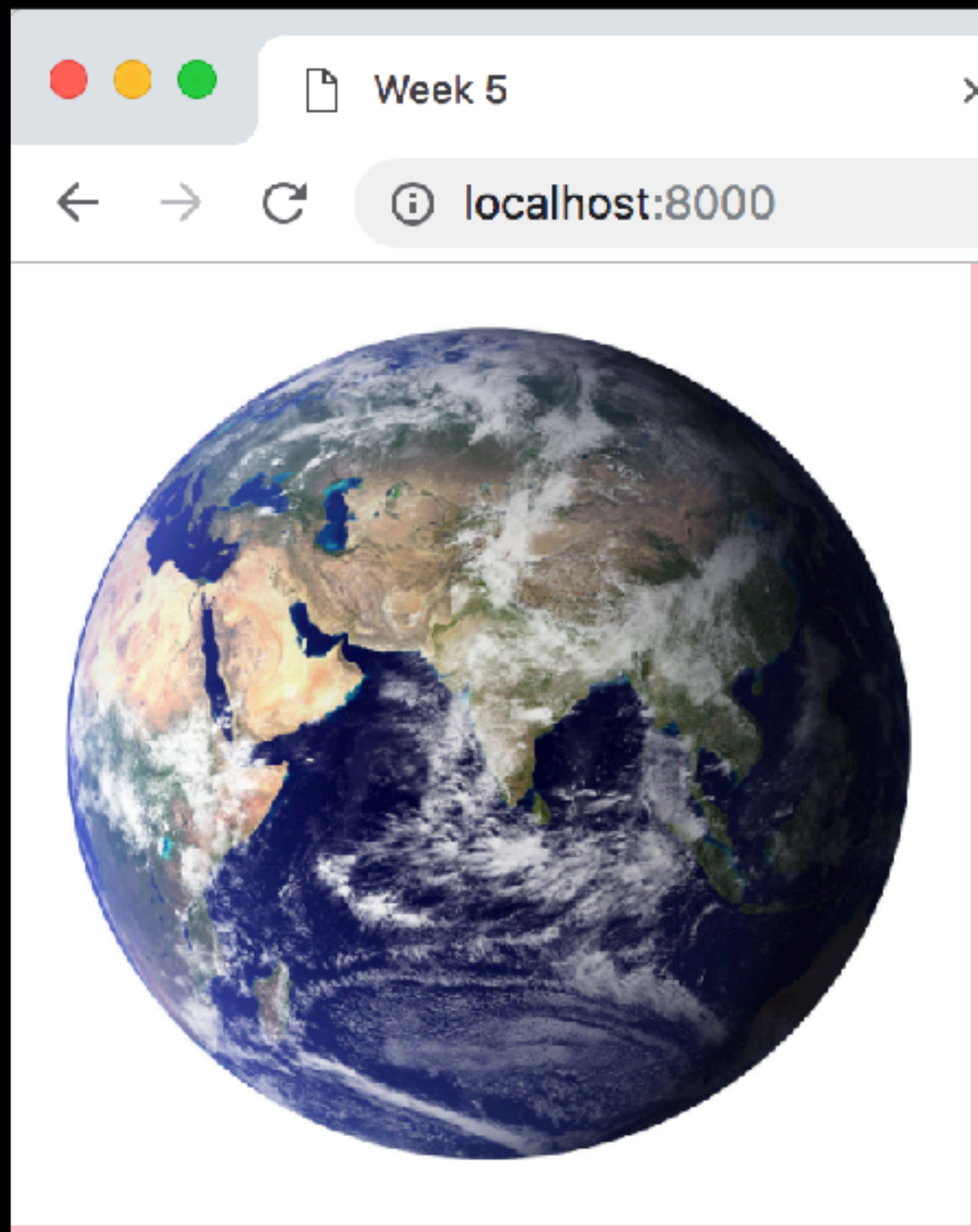

```
<img src="globe.png" width="300">
```

JPG

Joint Photographic Networks Group

a commonly used method of **lossy compression** for digital images, particularly for those images produced by digital photography. The degree of compression can be adjusted, allowing a selectable tradeoff between storage size and image quality.

.jpg files have no alpha channel.




```
<img src="globe.jpg" width="300">
```

GIF

Graphic Interchange Format

Steve Whilhite

1987, CompuServe