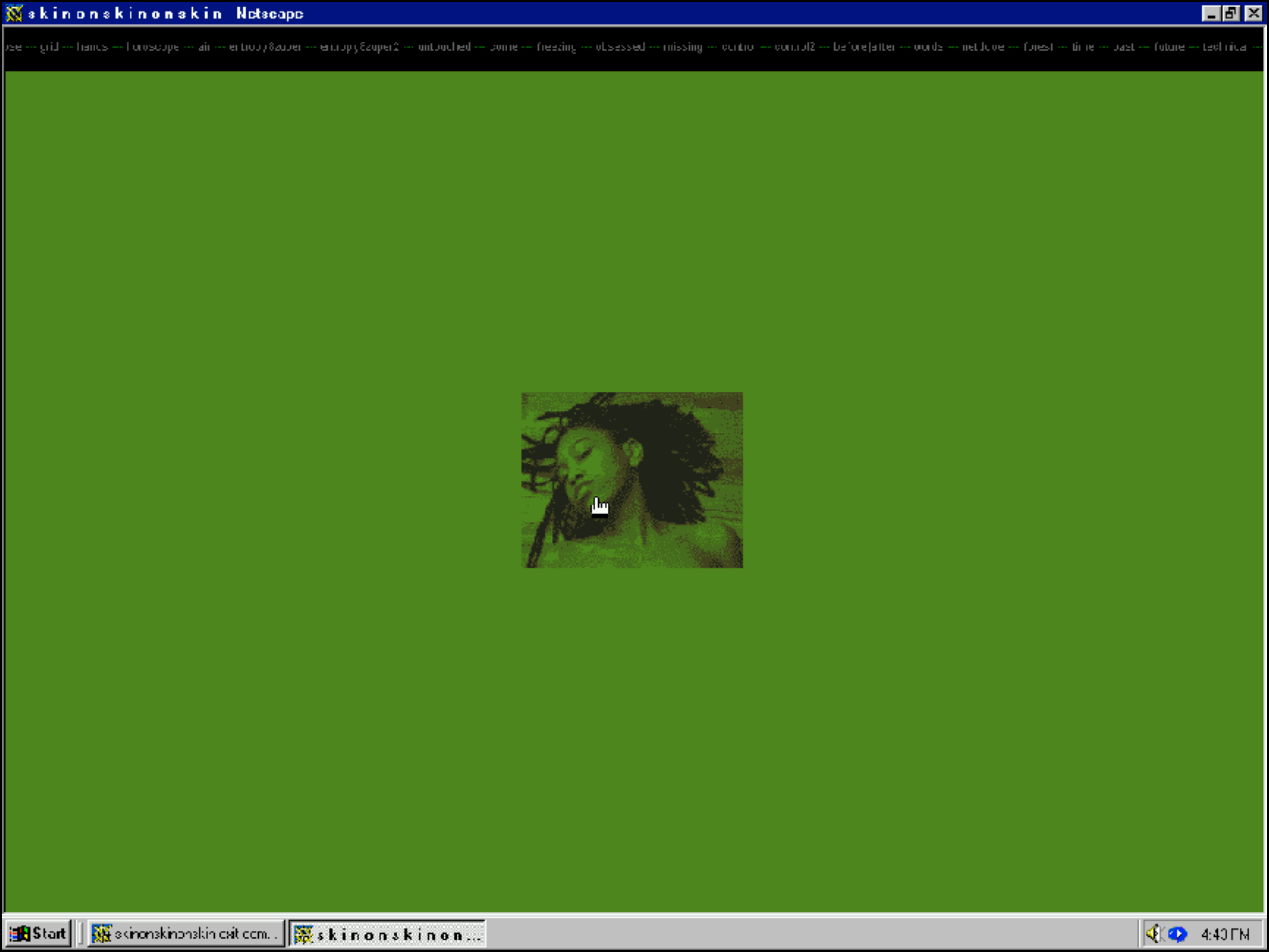

skinonskinonskin Netscape
grid --- hands --- horoscope --- air --- entropy8zuper --- entropy8zuper2 --- untouched --- some --- freezing --- obsessed --- missing --- control --- control2 --- before|after --- words --- netlove --- forest --- time --- past --- future --- technica
Start
skinonskinonskin exit com...
skinonskinon...
4:40 PM

skinonskinonskin, 1999
Auriea Harvey (Entropy8) + Michaël Samyn (Zuper!)
Entropy8Zuper!

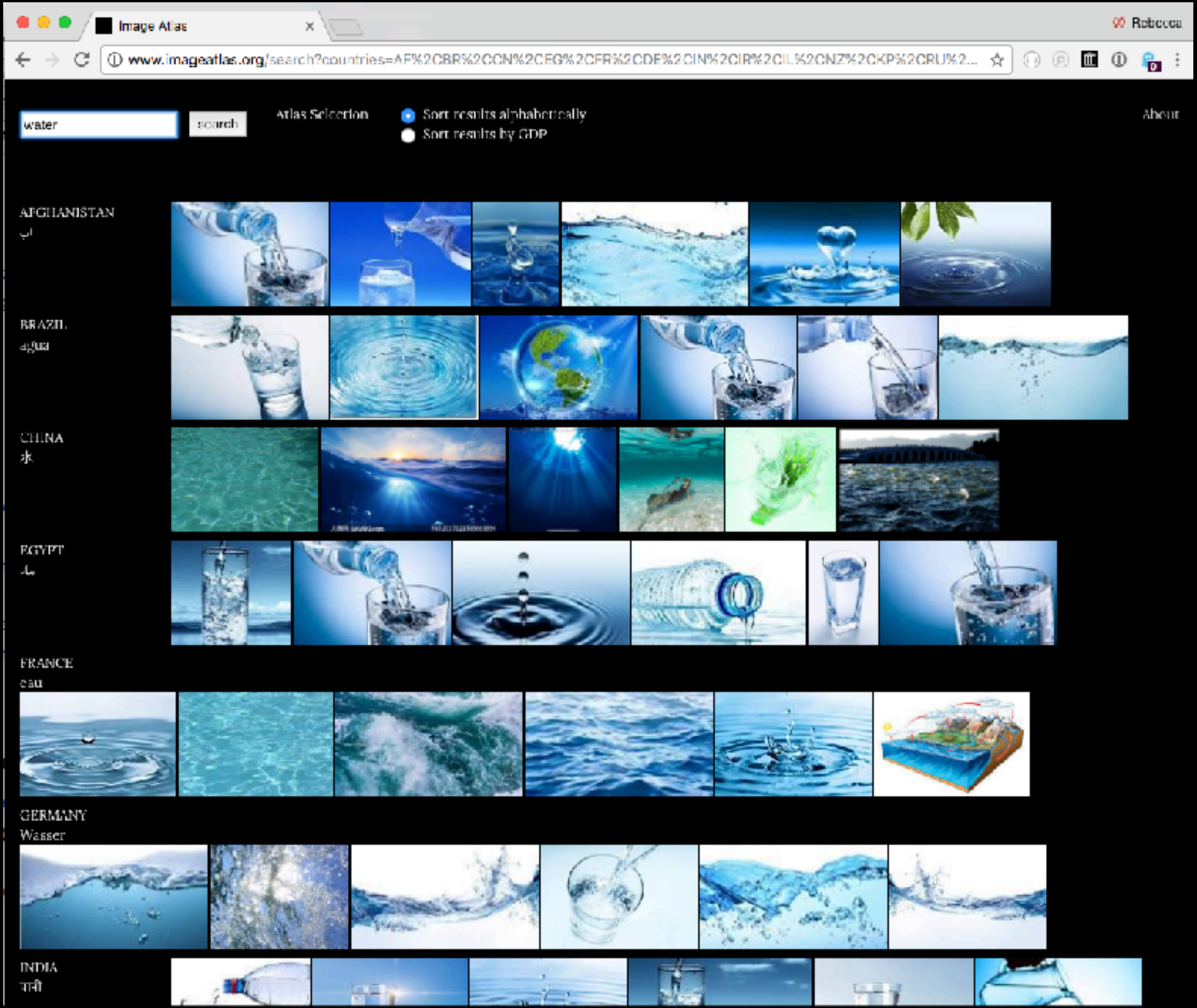
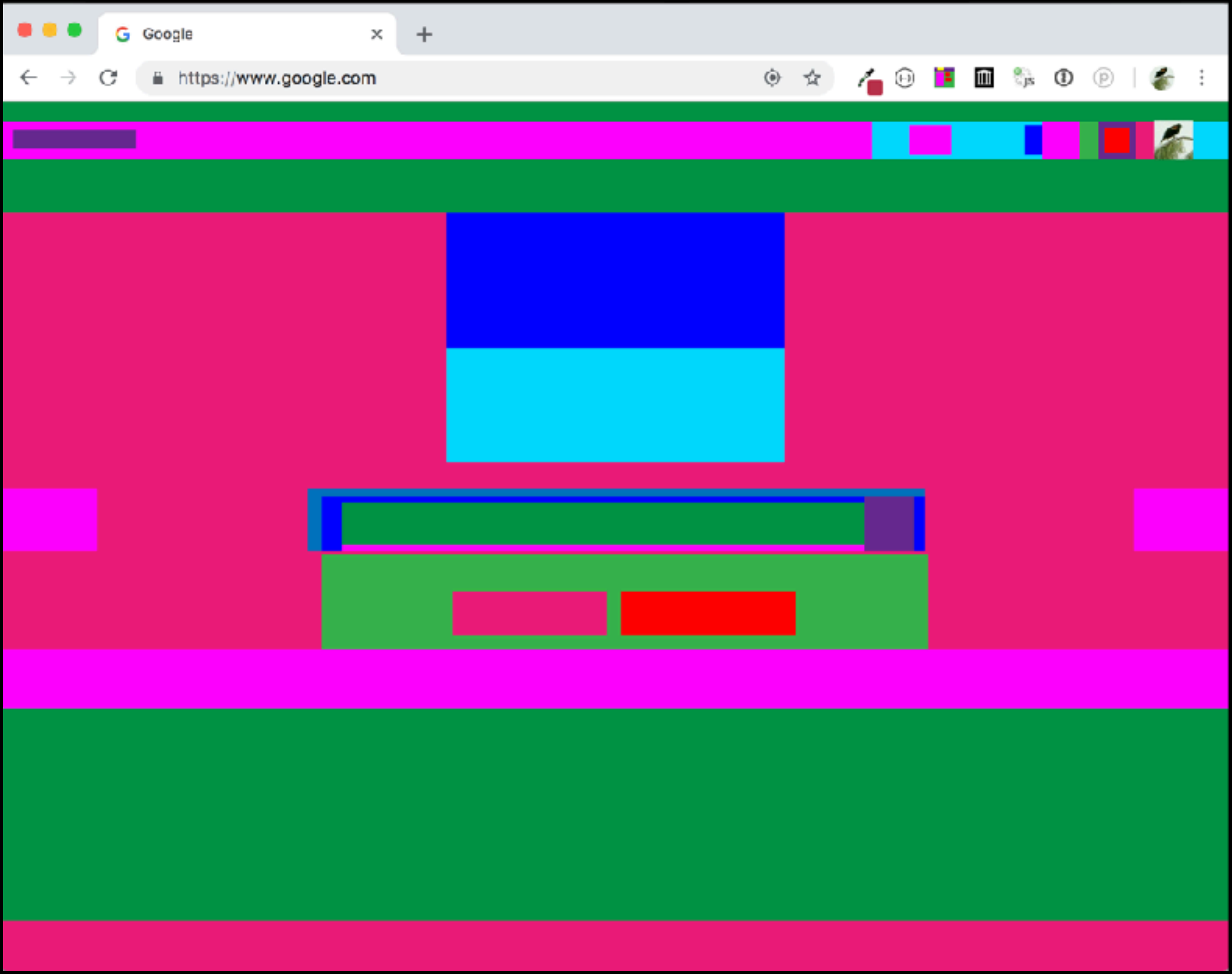
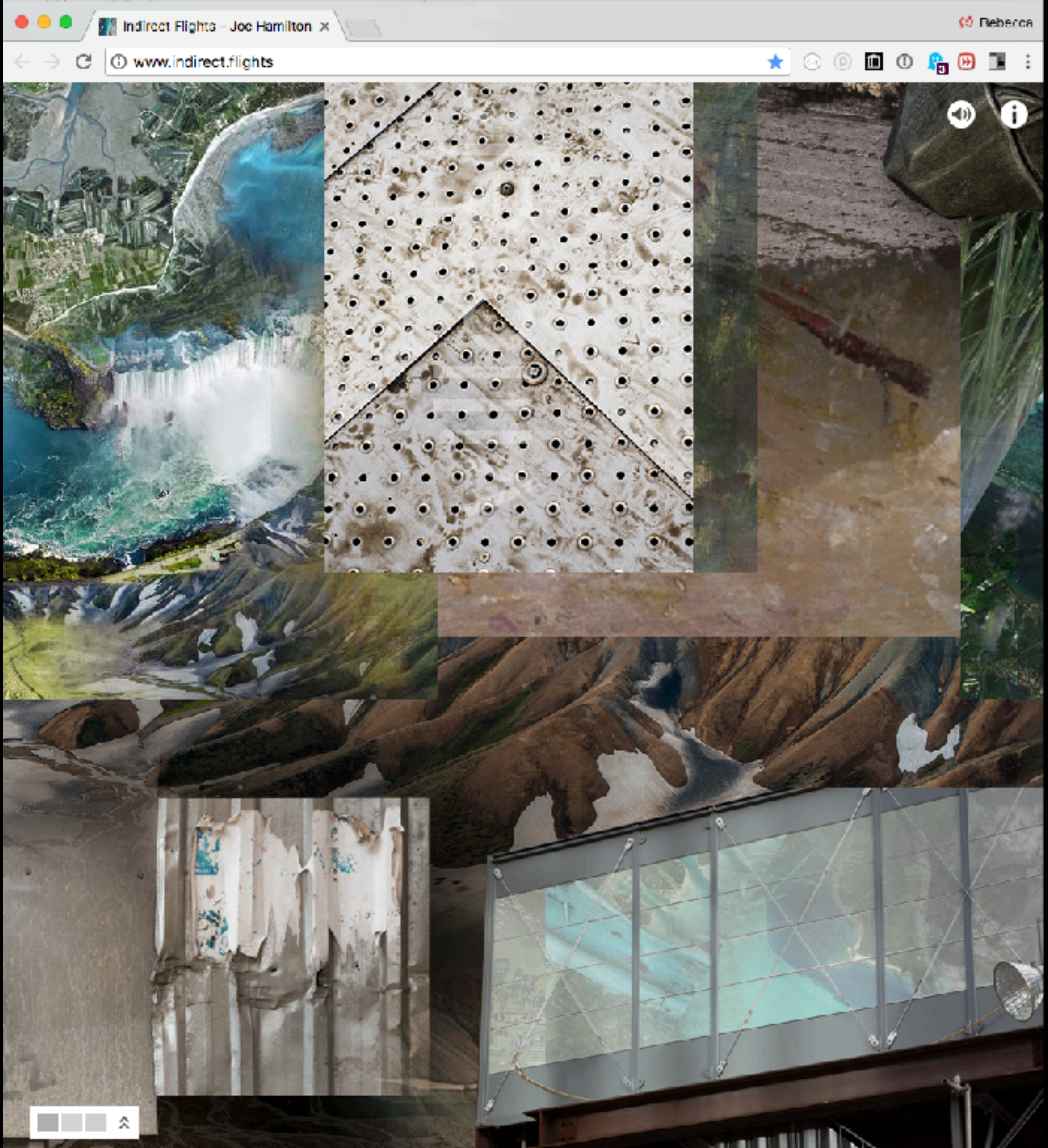


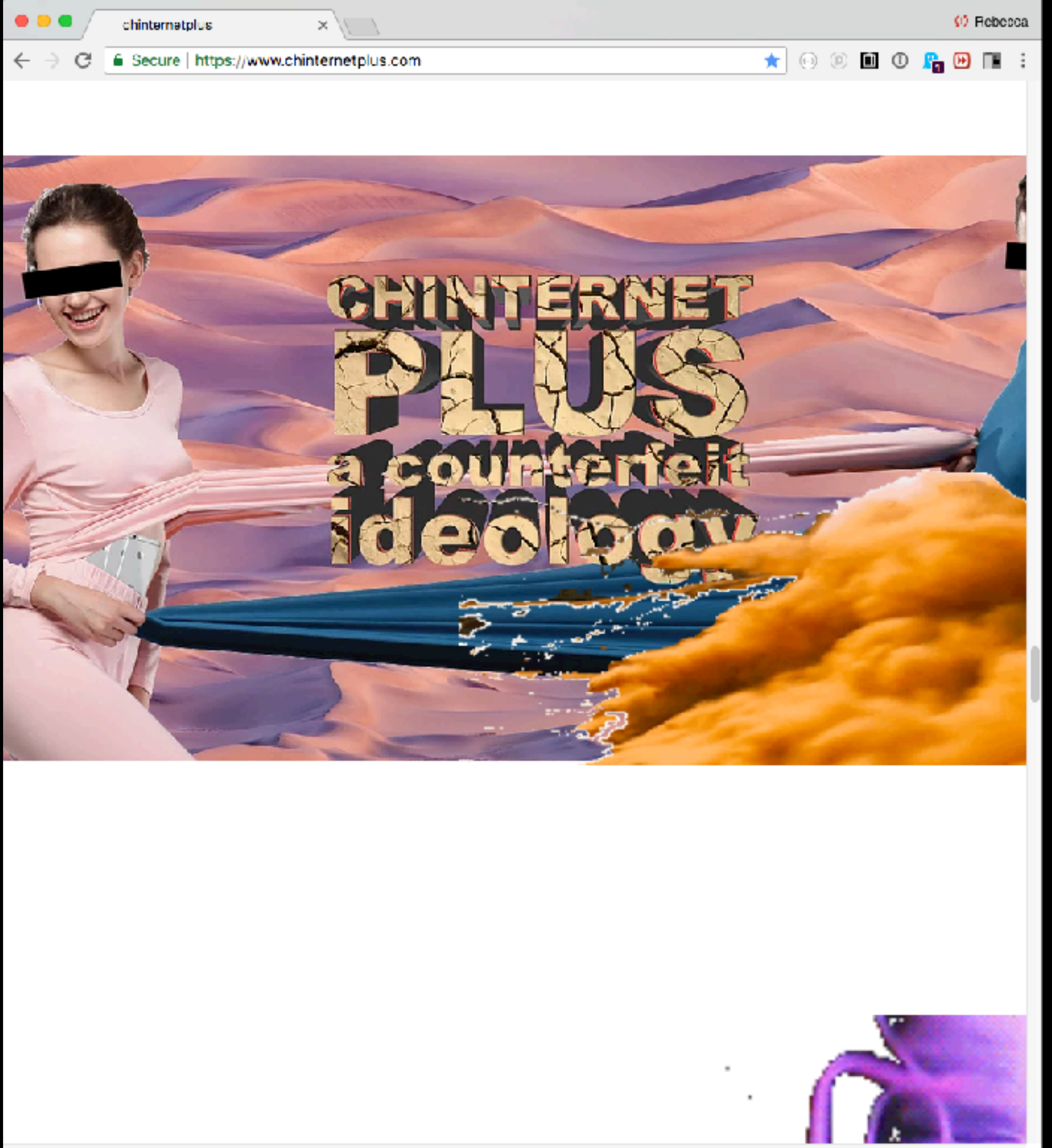
Image Atlas, 2012
Taryn Simon + Aaron Swartz



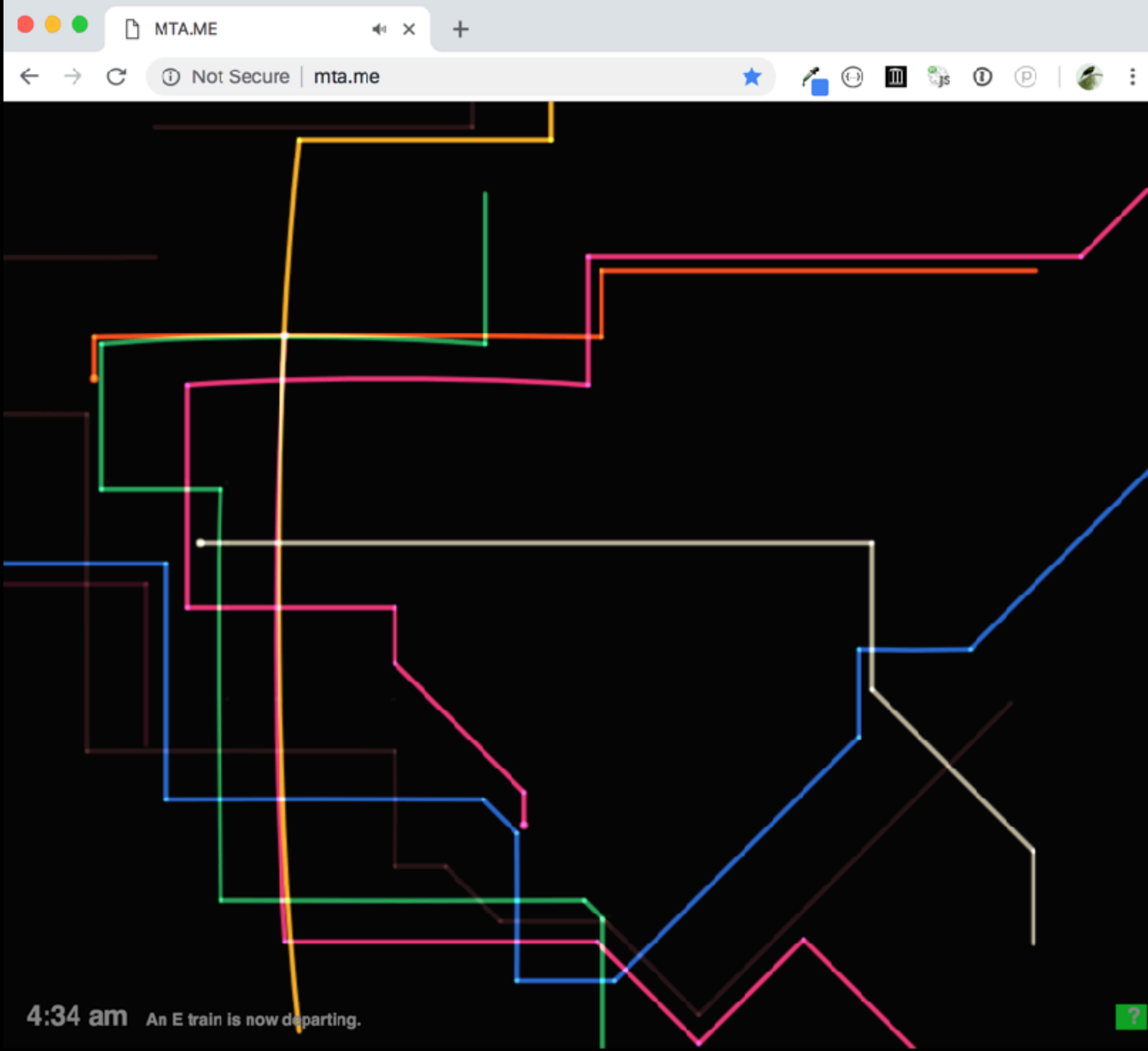
Rafaël Rozendaal.
Abstract Browsing, 2014



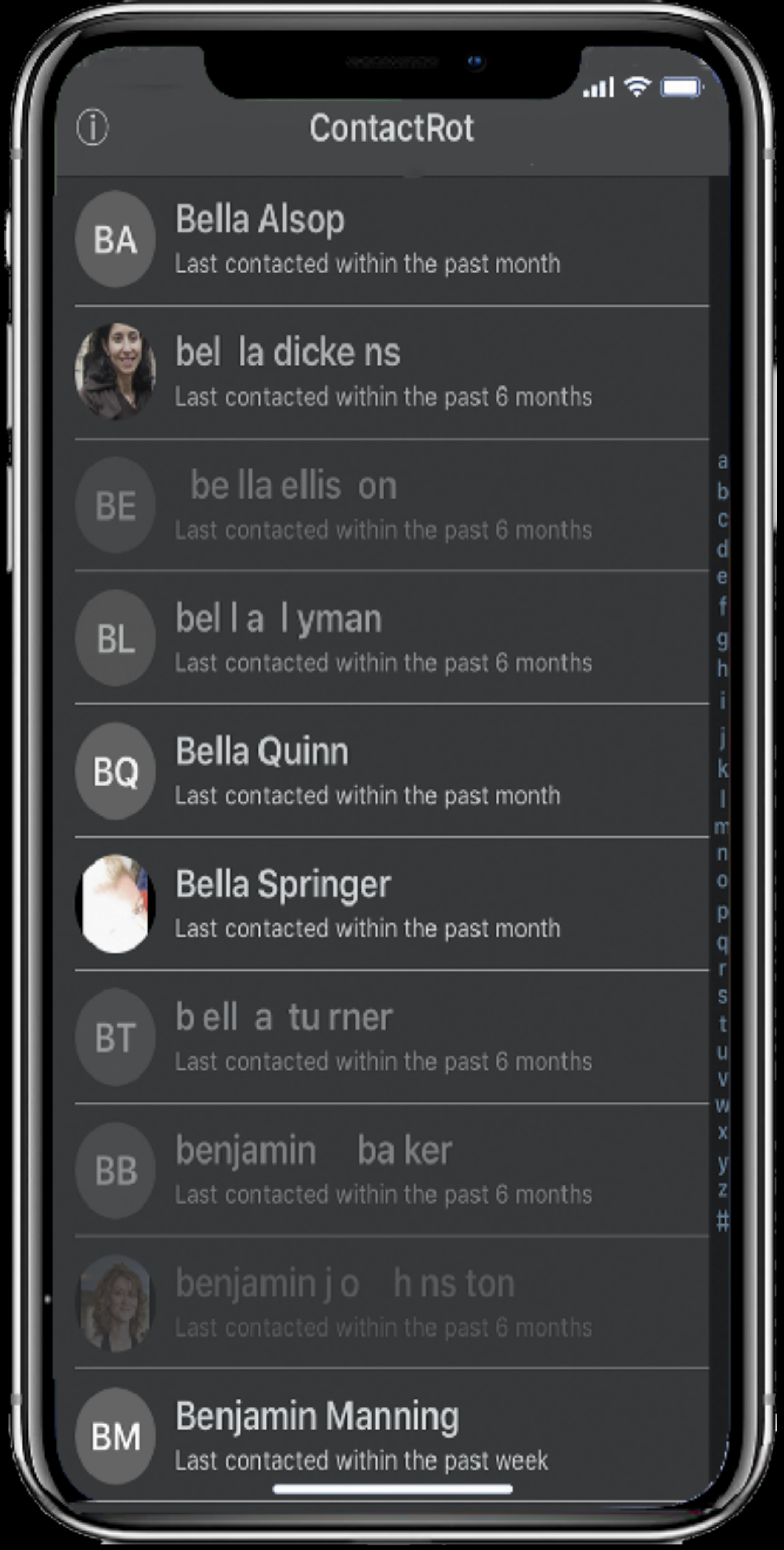
Indirect Flicts, 2015
Joe Hamilton



Chinternet Plus, 2016
Miao Ying



MTA.ME, 2011
Alexander Chen



ContactRot

BA Bella Alsop
Last contacted within the past month

bel la dicke ns
Last contacted within the past 6 months

BE be lla ellis on
Last contacted within the past 6 months

BL bel l a l yman
Last contacted within the past 6 months

BQ Bella Quinn
Last contacted within the past month

Bella Springer
Last contacted within the past month

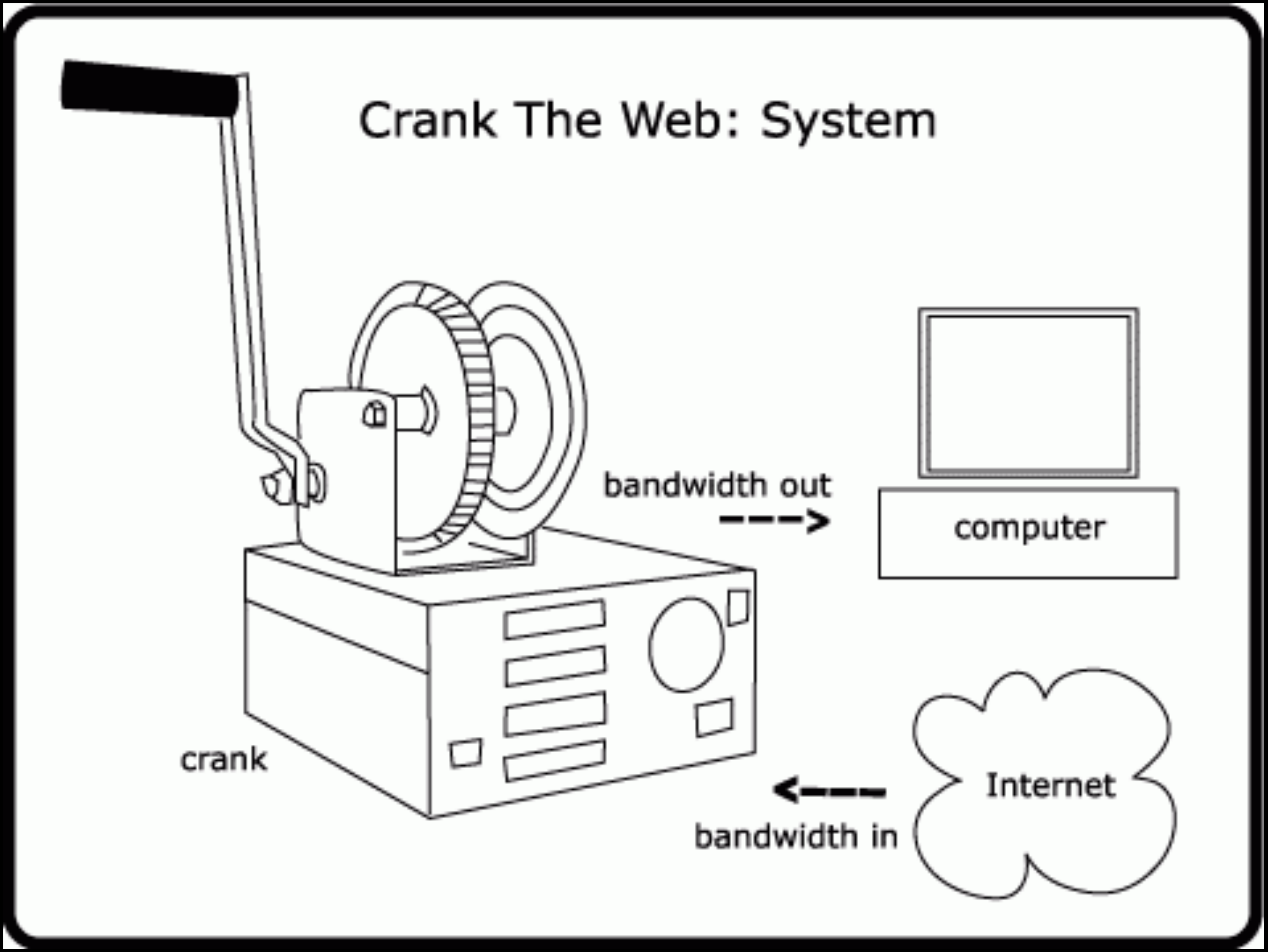
BT b ell a tu rner
Last contacted within the past 6 months

BB benjamin   ba ker
Last contacted within the past 6 months

benjamin j o   h ns ton
Last contacted within the past 6 months

BM Benjamin Manning
Last contacted within the past week

Jonah Brucker-Cohen
Contact Rot, 2018



Jonah Brucker-Cohen
Crank the Web, 2001

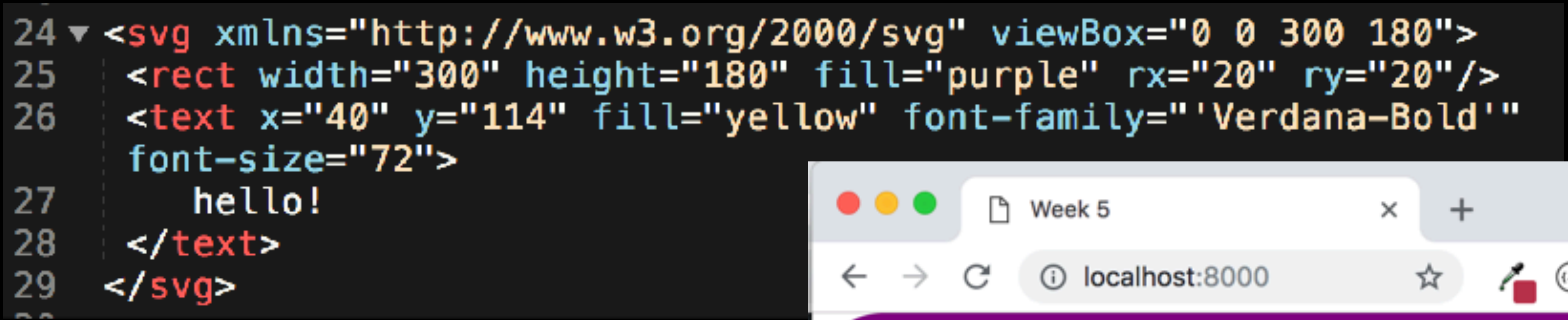
image files for the web

SVG

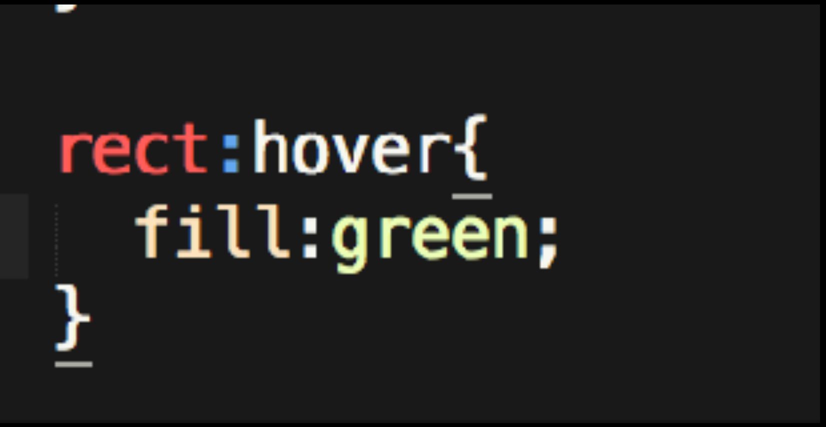
SVG is a 2D vector imaged based on an **XML** (Extensible Markup Language) syntax. It provides the rules and standards for how markup languages should be written and work together. As a result, SVG works well with HTML content.

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)**.

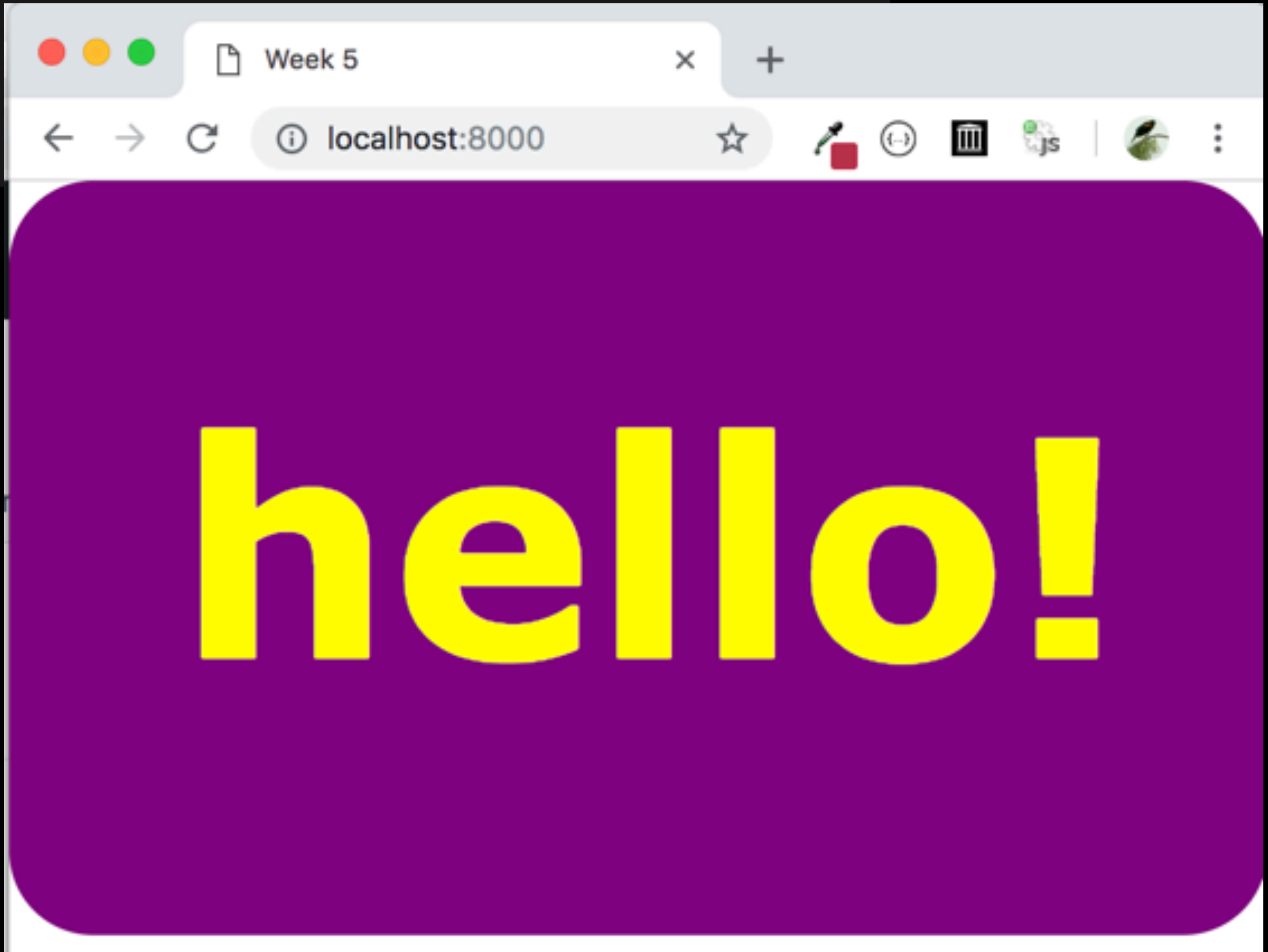
A simple example will give you the general idea. Here is the SVG code that describes a rectangle **(rect)** with rounded corners (rx and ry, for x-radius and y-radius) and the word "hello" set as text with attributes for the font and color. Browsers that support SVG read the instructions and draw the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



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.

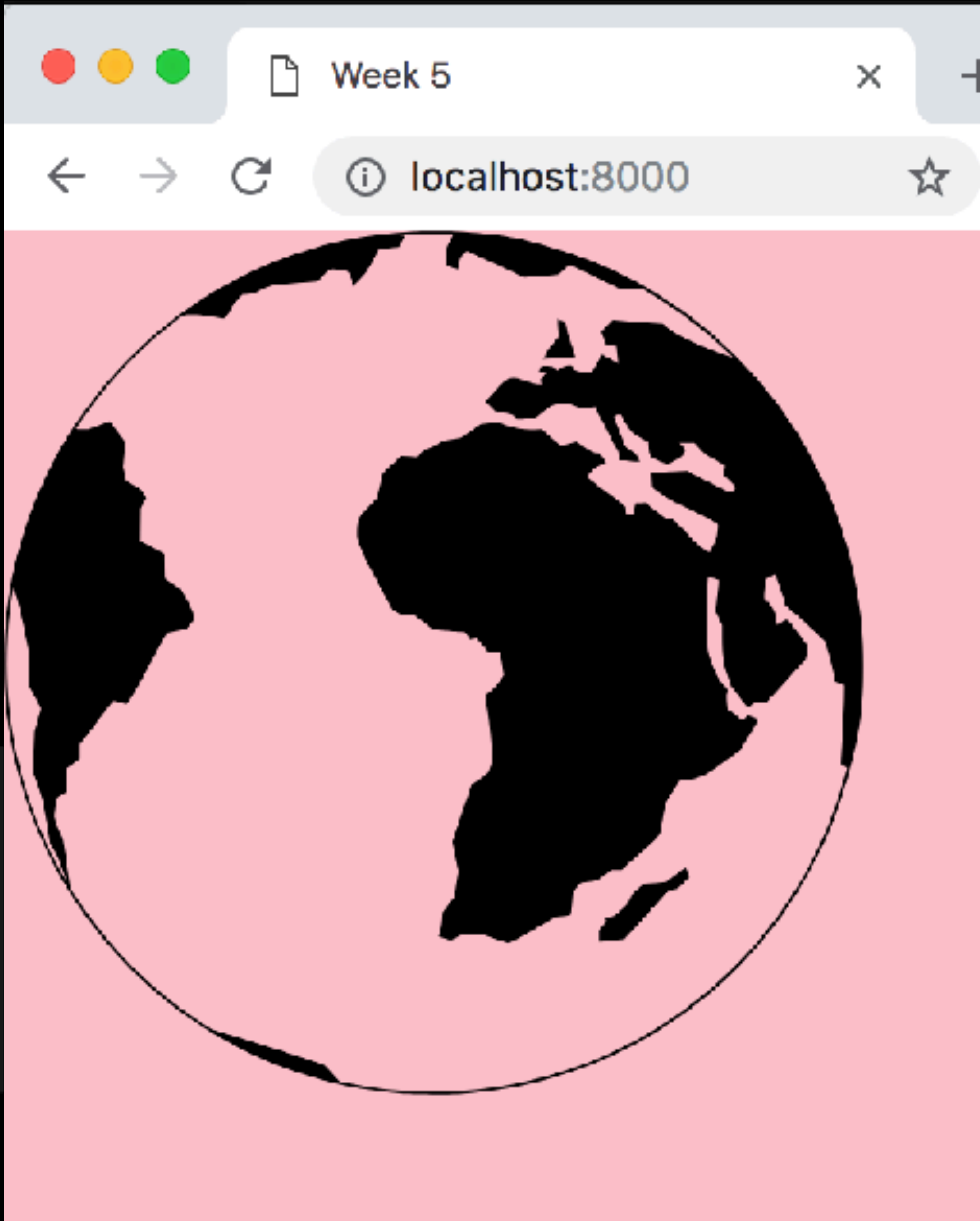
You can change how they look with CSS.

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

SVG

Embedded with the img element - this will act as a static image.
You can not apply styles, animation or javascript:

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



Metadata: `viewport`

The user's visible area of a web page

HTML5 introduced a method to let web designers take control over the viewport, through the **<meta>** tag.

```
<!
—  - Tells the browser to match the device's width for the viewport
   - Sets an initial zoom value -->

<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;

responsive text

The text size can be set with a "vw" unit, which means the "viewport width".

That way the text size will follow the size of the browser window.

```
<h1 style="font-size:10vw">Hello World</h1>
```

rwd
Responsive Web Design

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

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**:

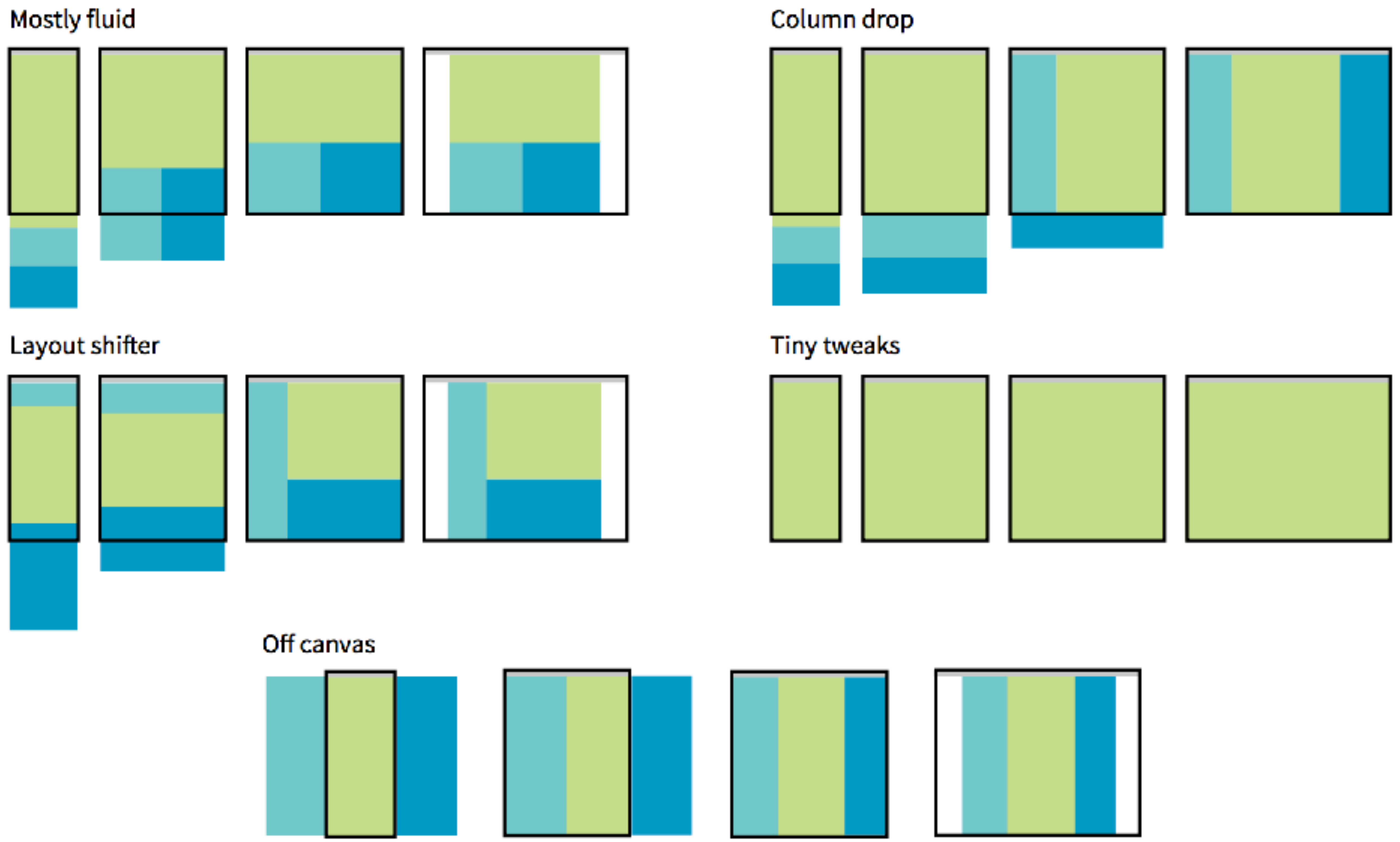


FIGURE 17-9. Examples of the responsive layout patterns identified by Luke Wroblewski.

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.

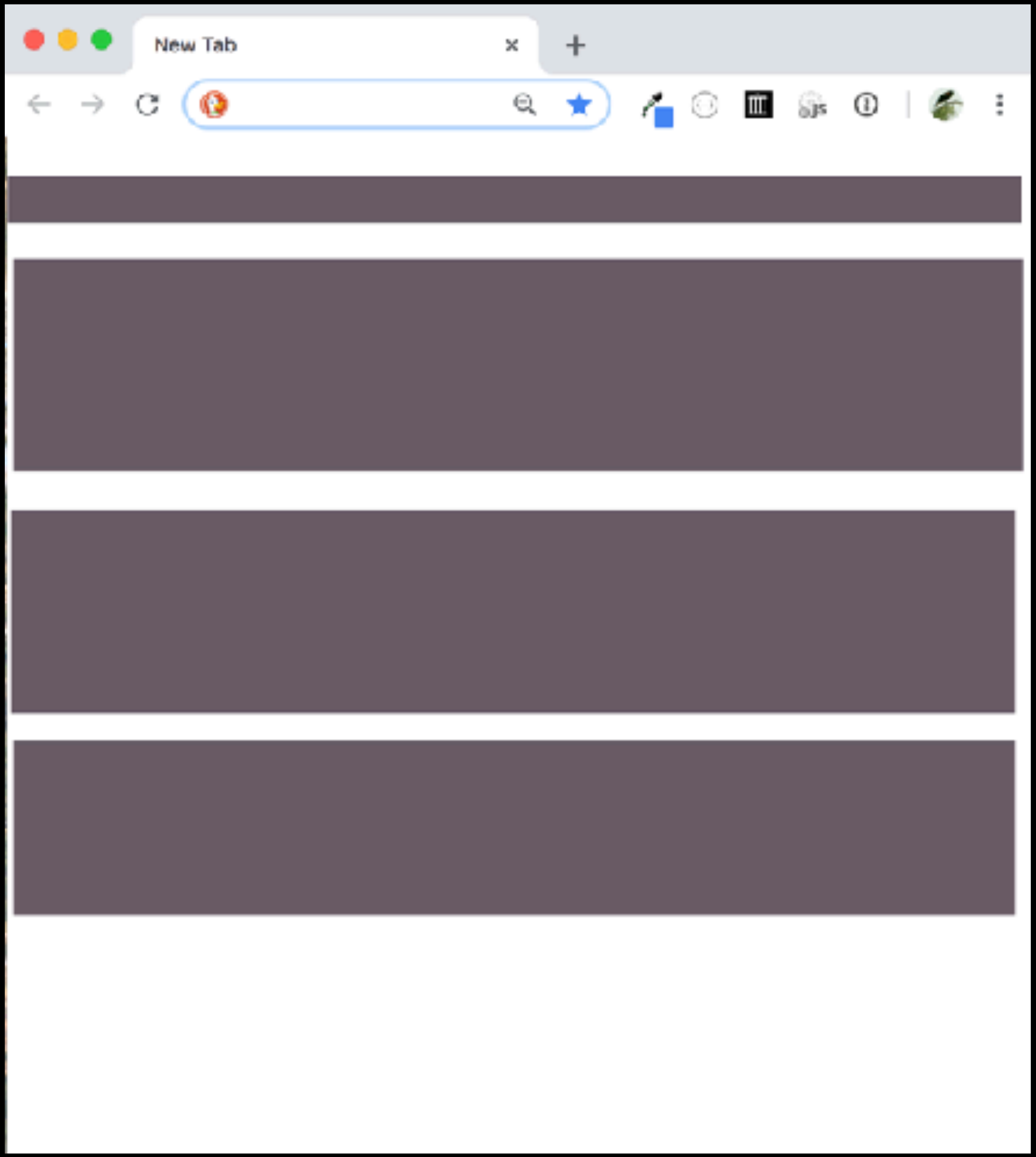
overflow-y

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;
}
```

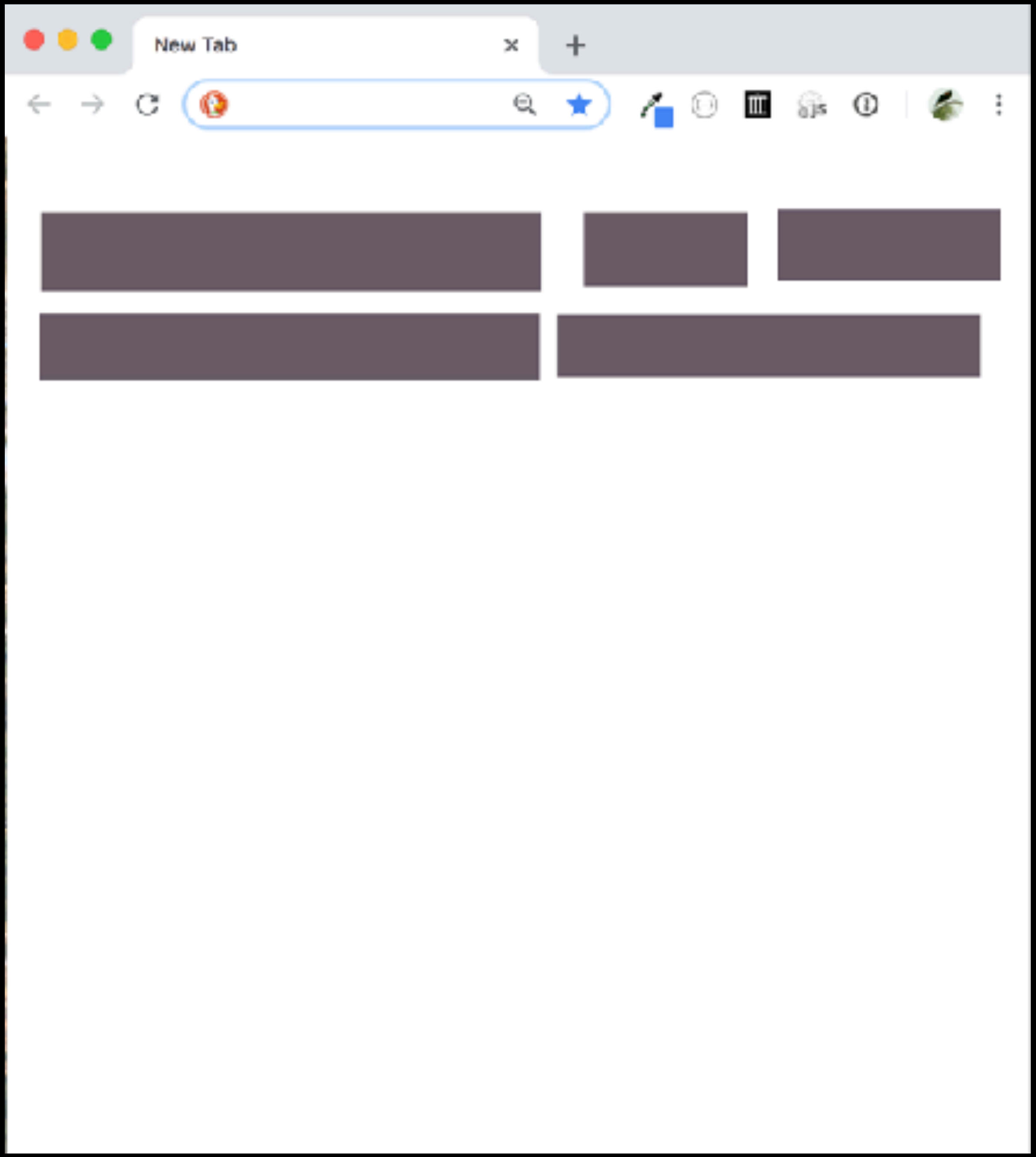
code from W3

flexbox



Block layout

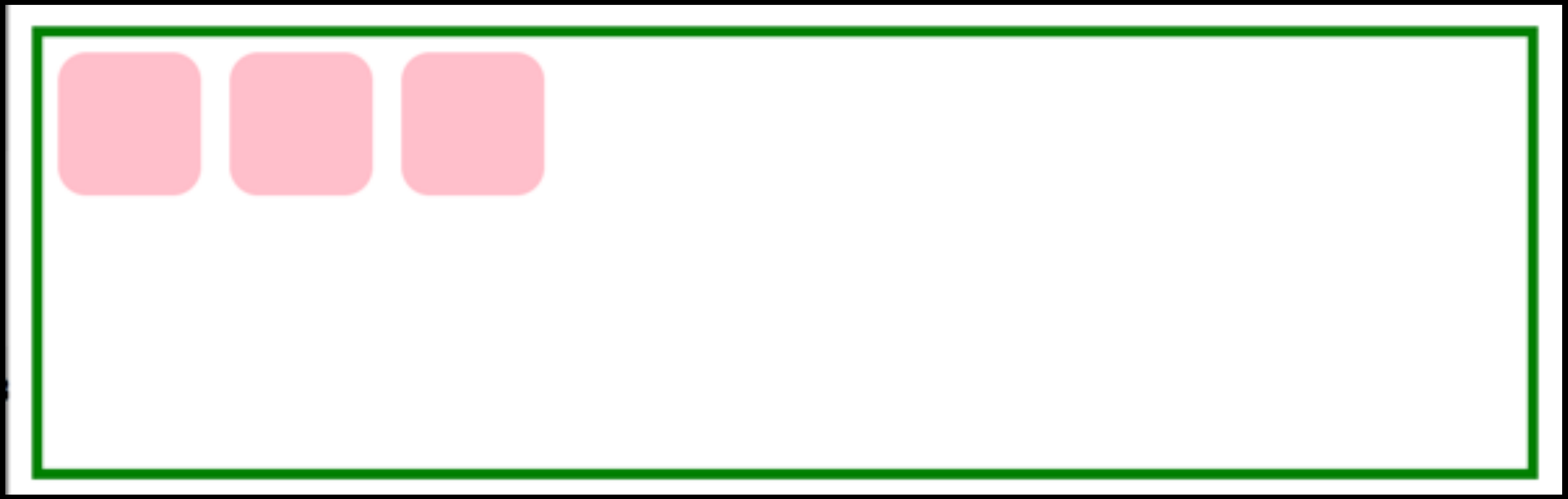
Laying out large sections of a page

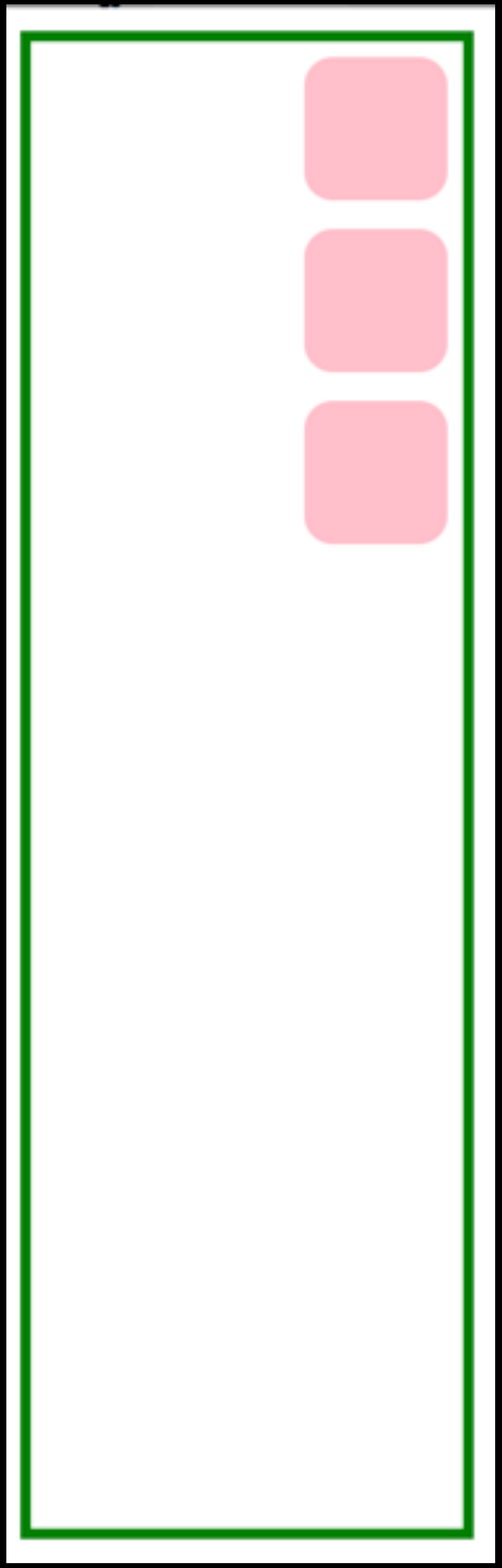


Inline layout

Laying out txt + other inline content within a section

Flex Layout

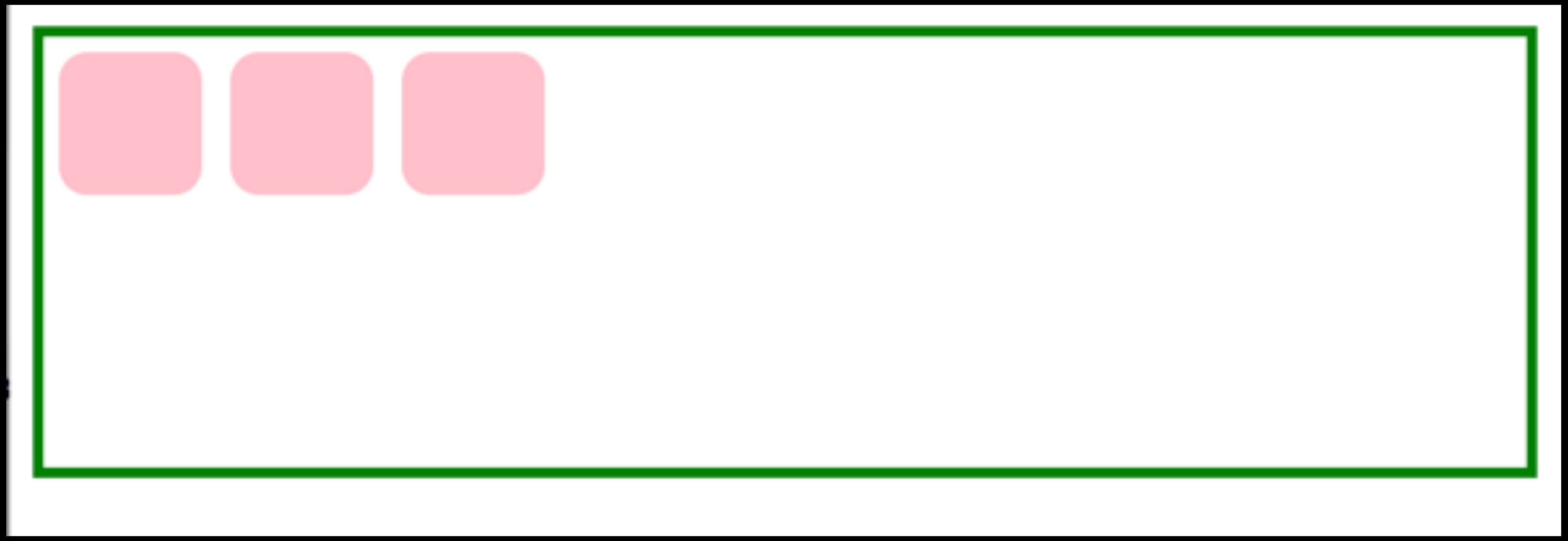




To achieve more complicated layouts, we can enable a different kind of CSS layout rendering mode: **Flex layout.**

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

Flex Basics



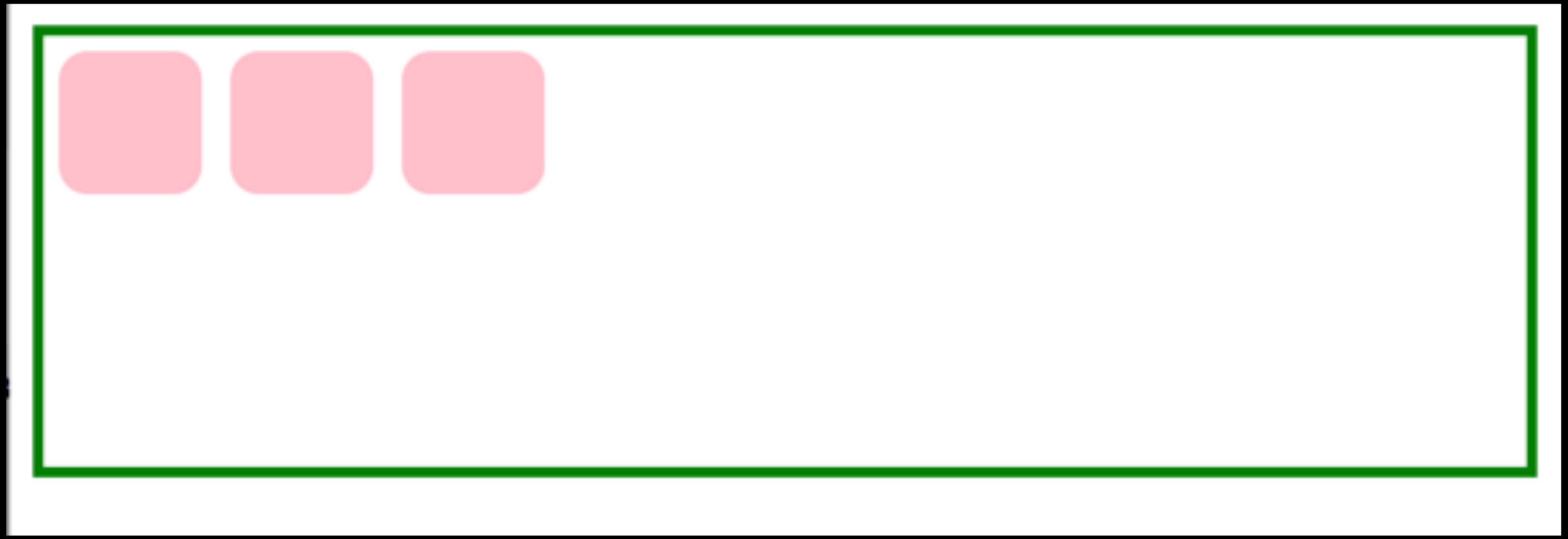
Flex layouts are composed of:

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

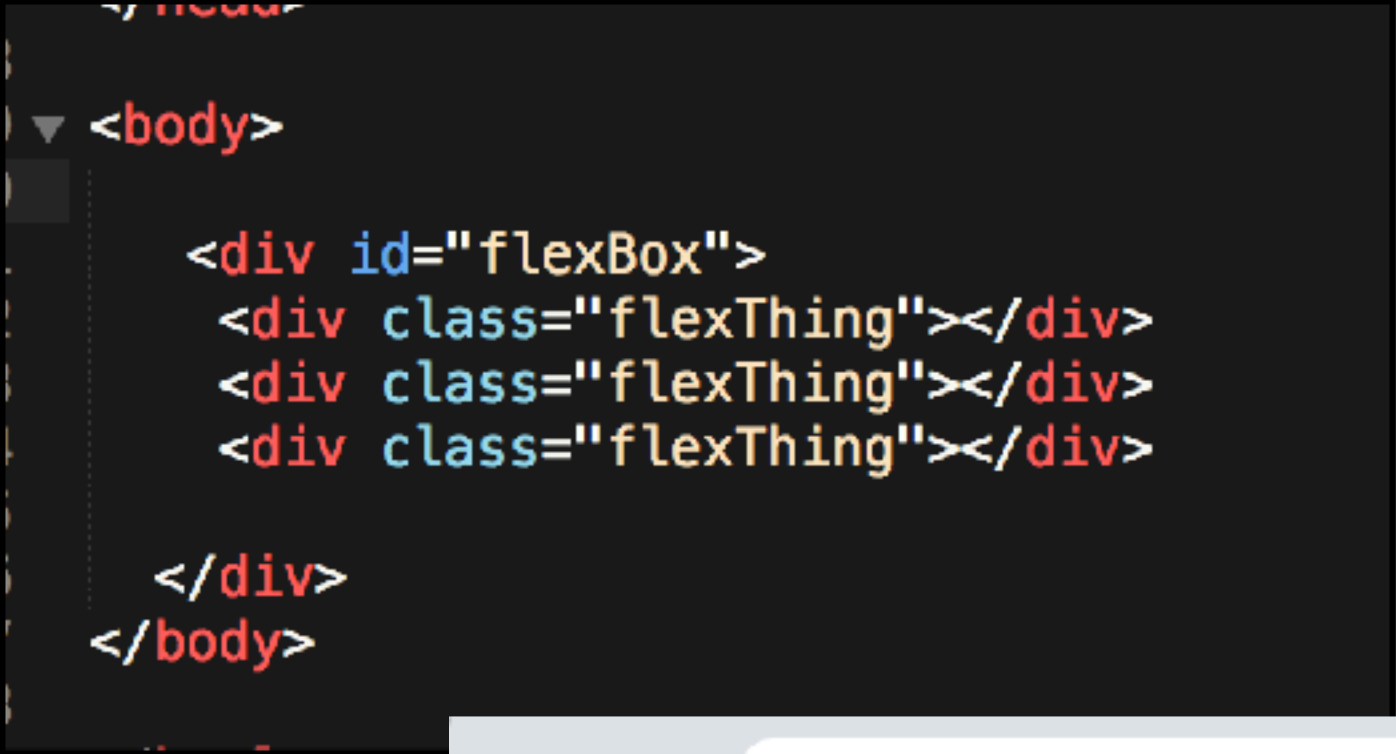
Flex Basics



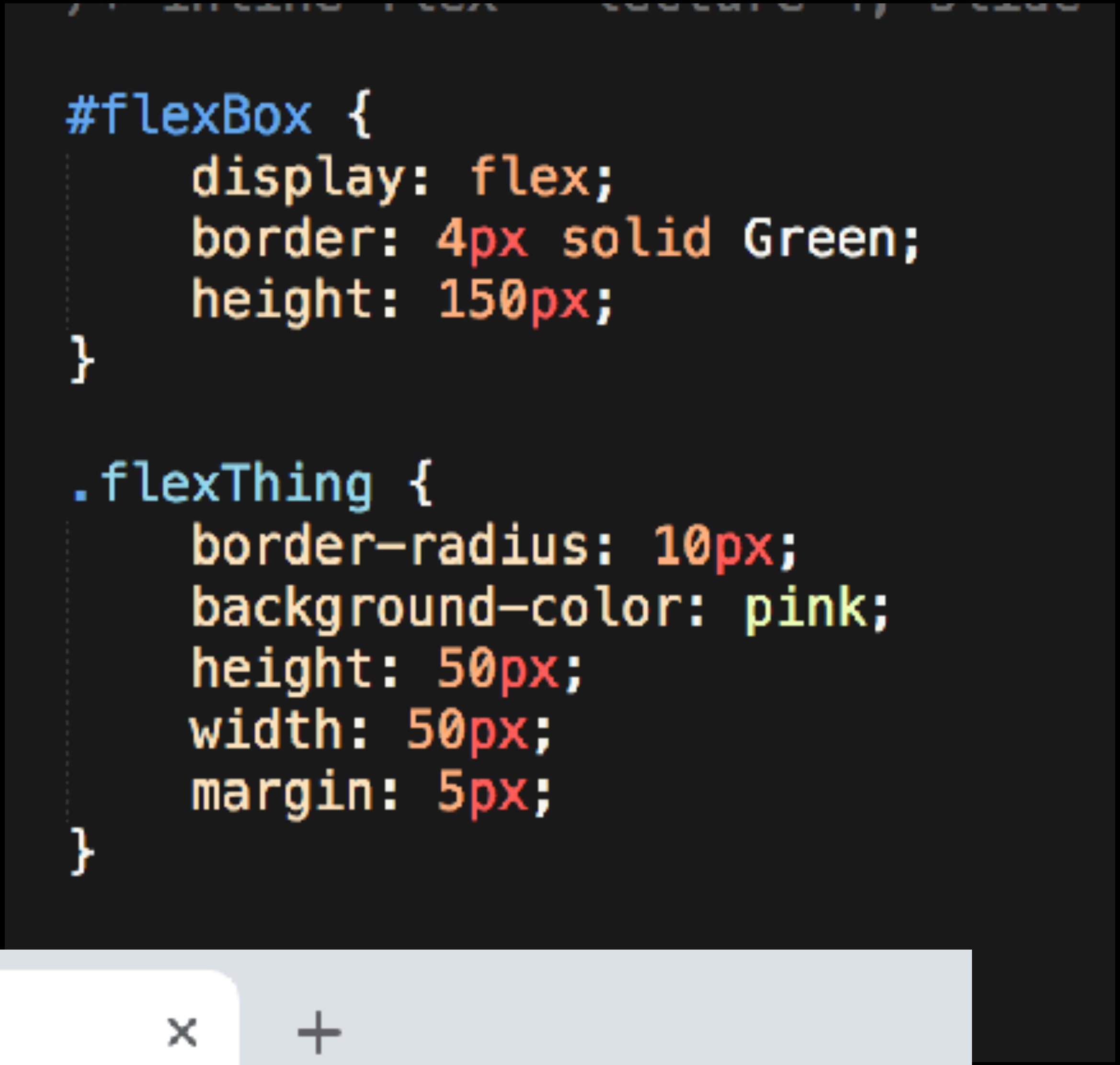
To make an element a flex container, change display:

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

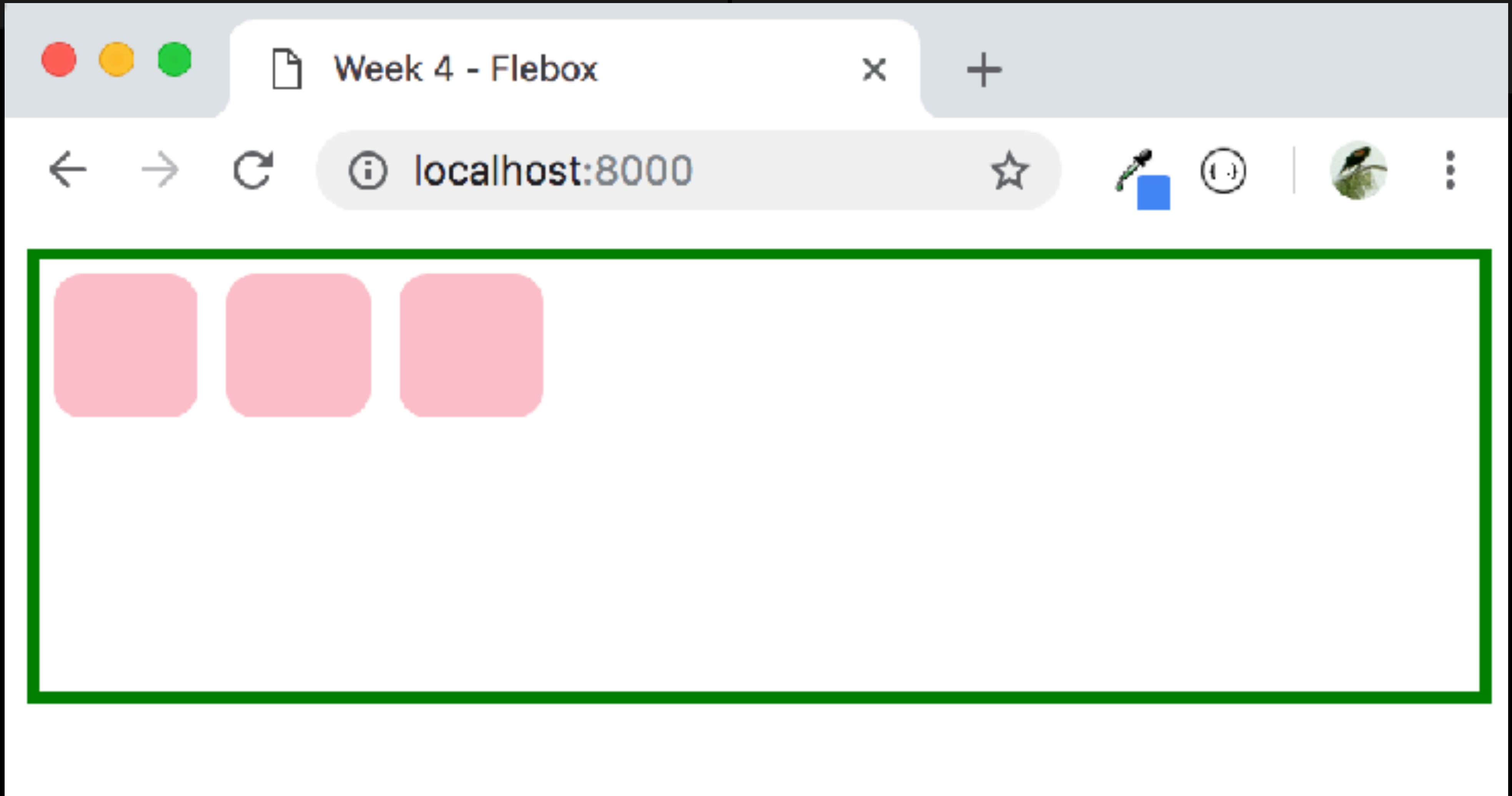
Flex Basics



```html
<body>

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

  </div>
</body>
```

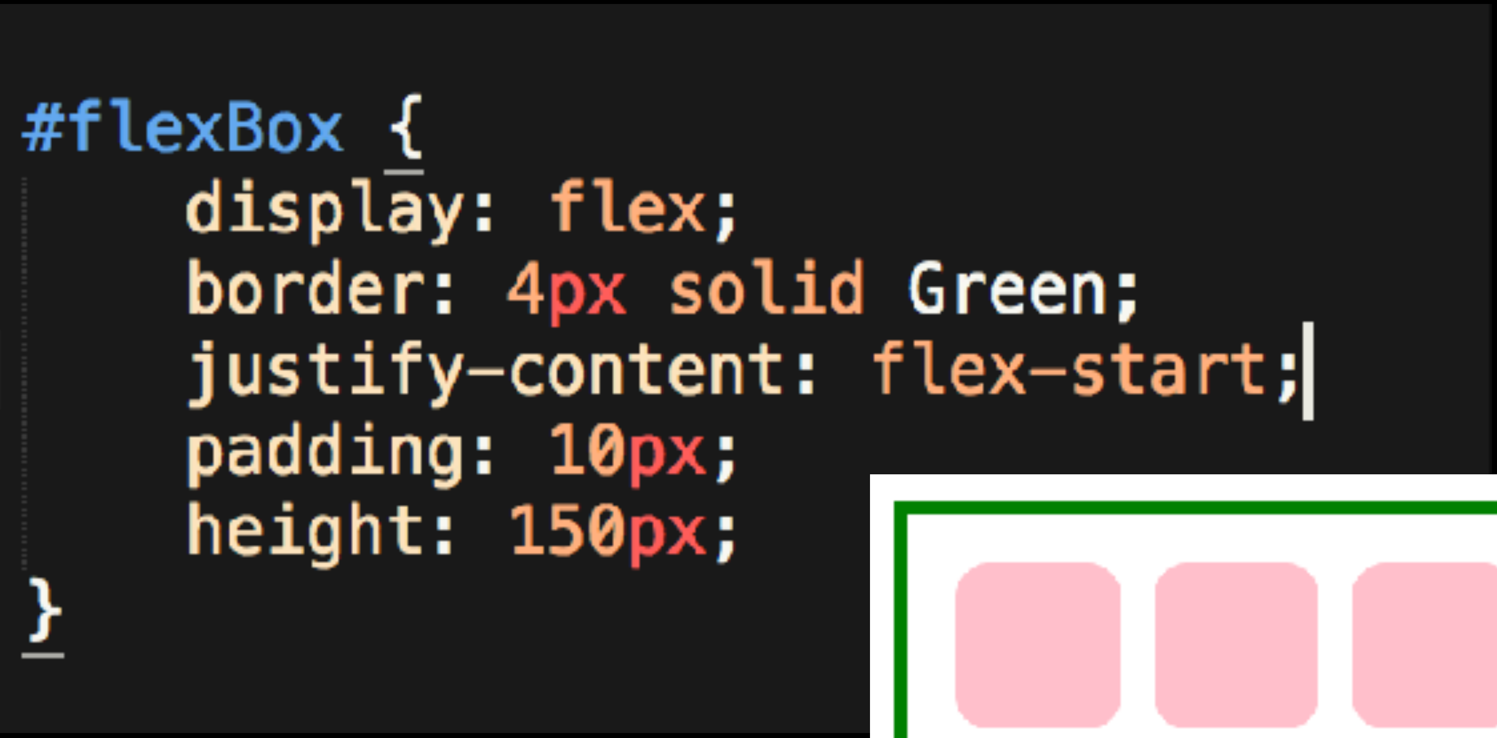


```css
#flexBox {
    display: flex;
    border: 4px solid Green;
    height: 150px;
}

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

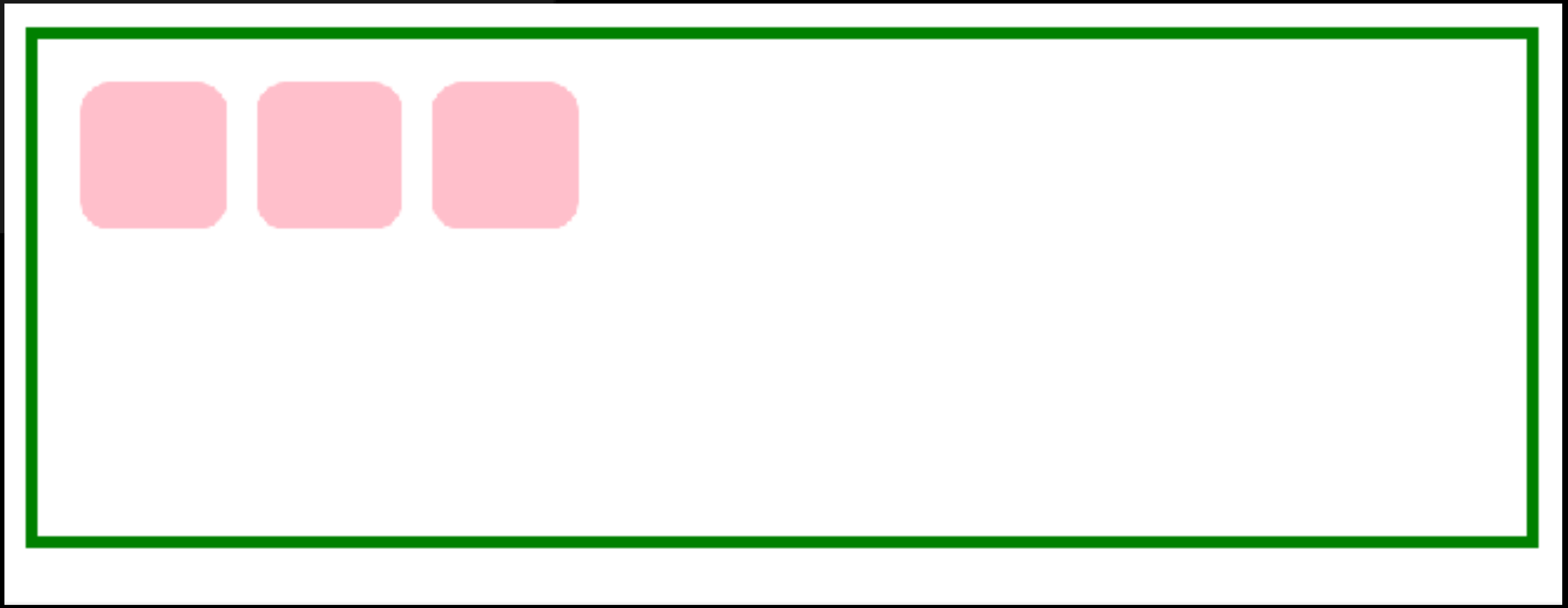


Flex Basics: justify-content

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

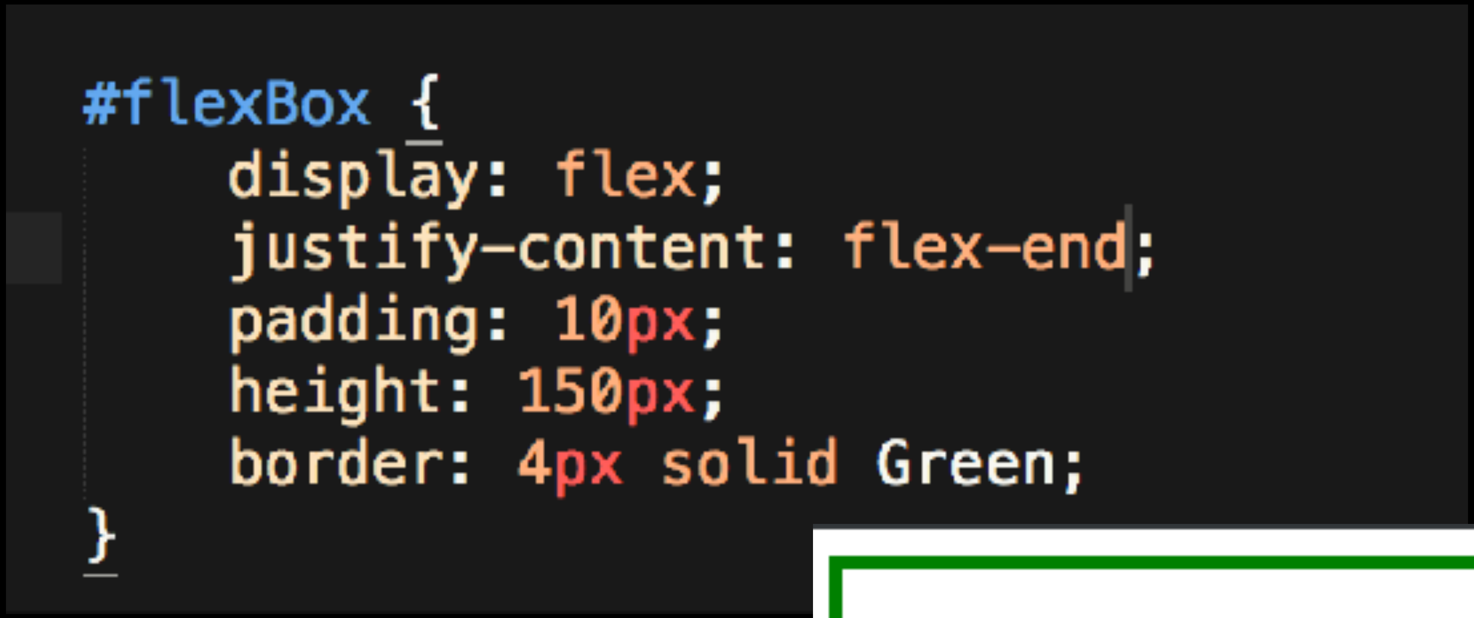


```
#flexBox {
    display: flex;
    border: 4px solid Green;
    justify-content: flex-start;
    padding: 10px;
    height: 150px;
}
```

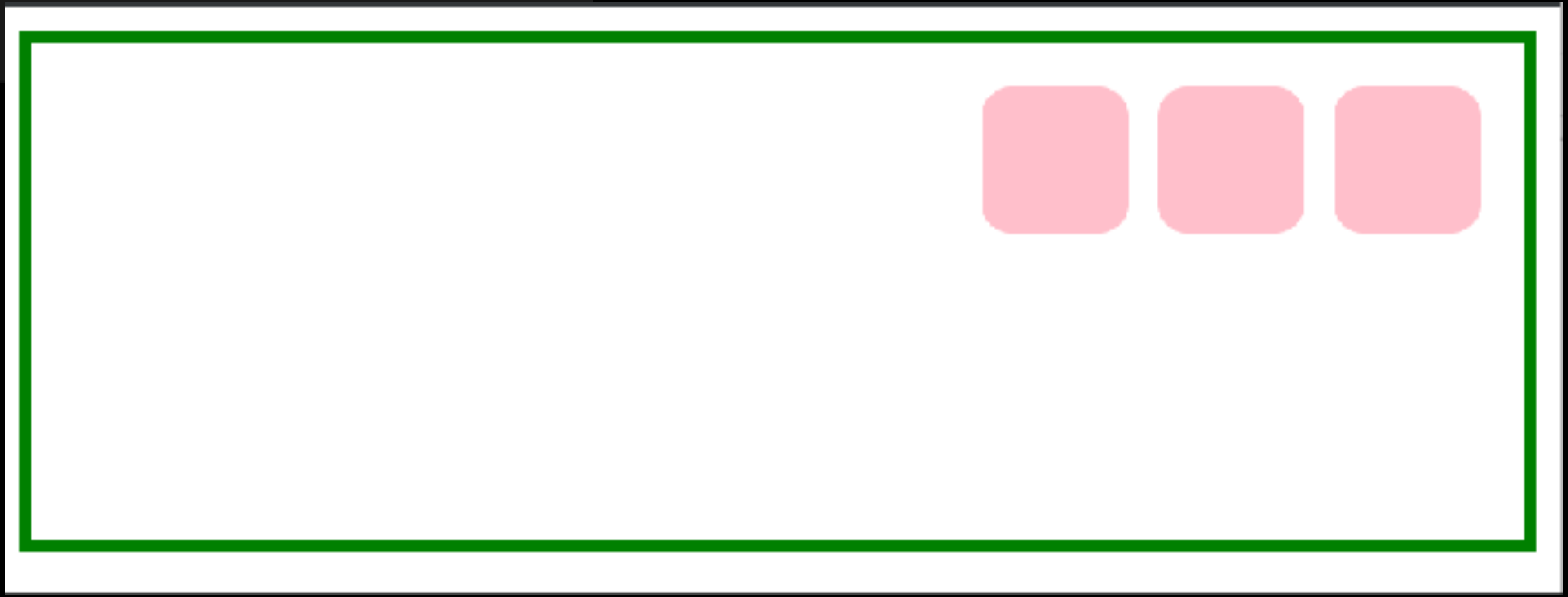


Flex Basics: justify-content

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

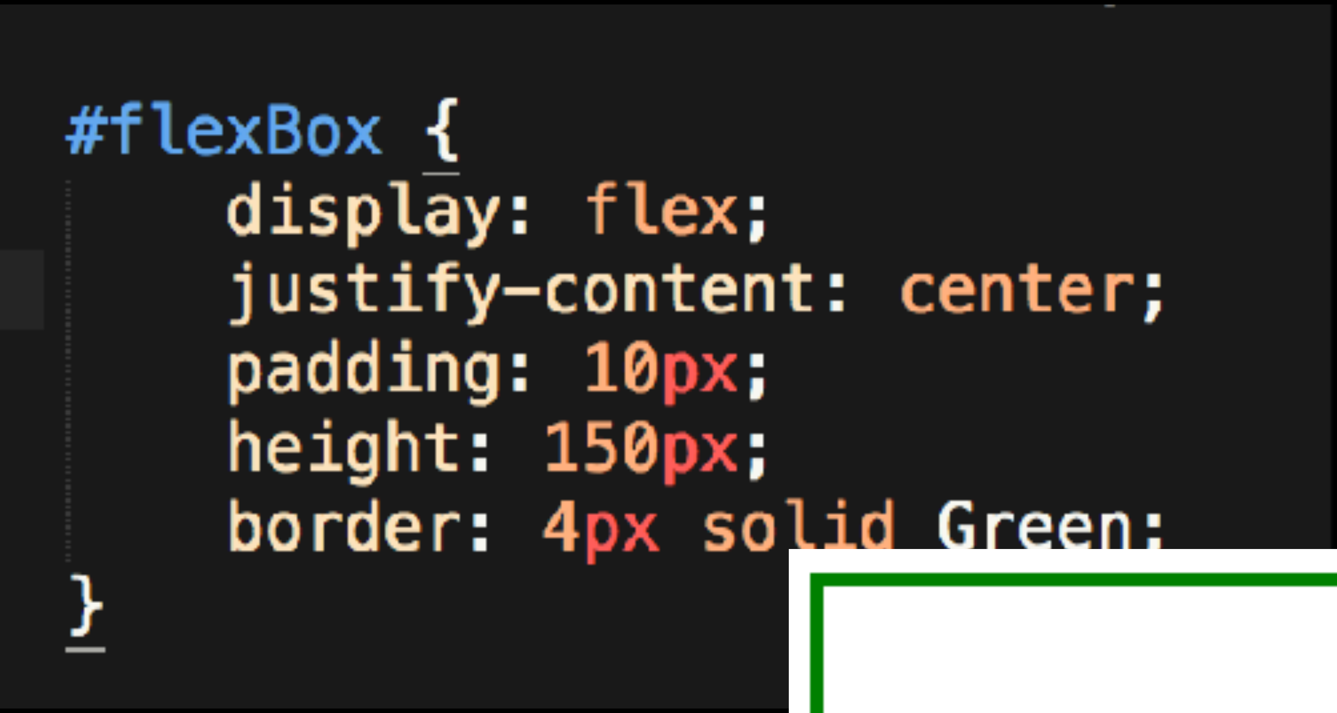


```
#flexBox {
    display: flex;
    justify-content: flex-end;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```

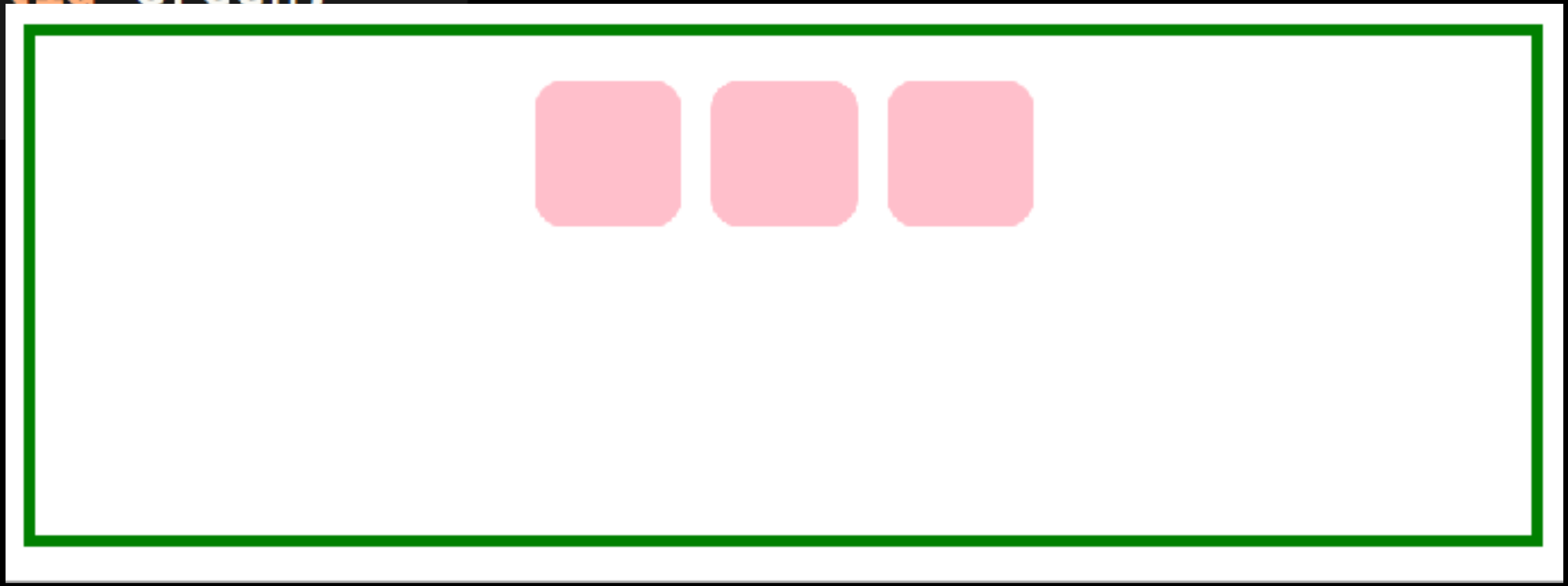


Flex Basics: justify-content

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

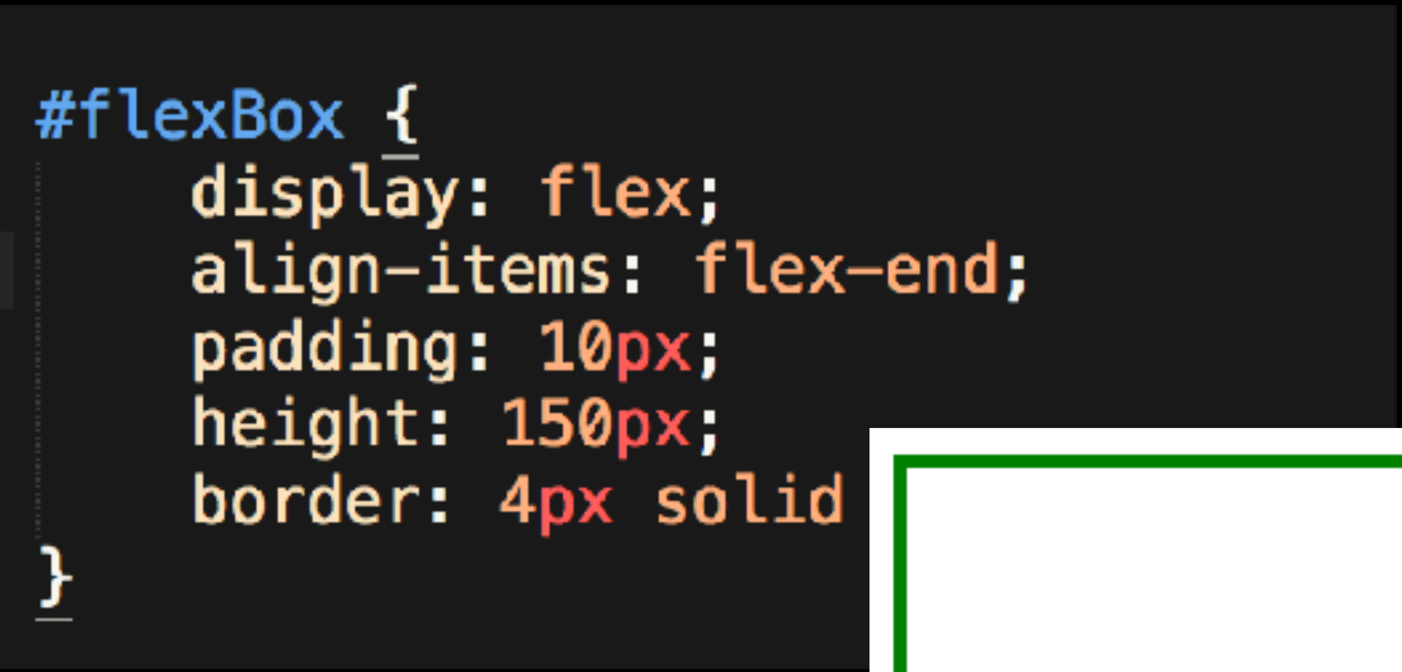


```
#flexBox {
    display: flex;
    justify-content: center;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```

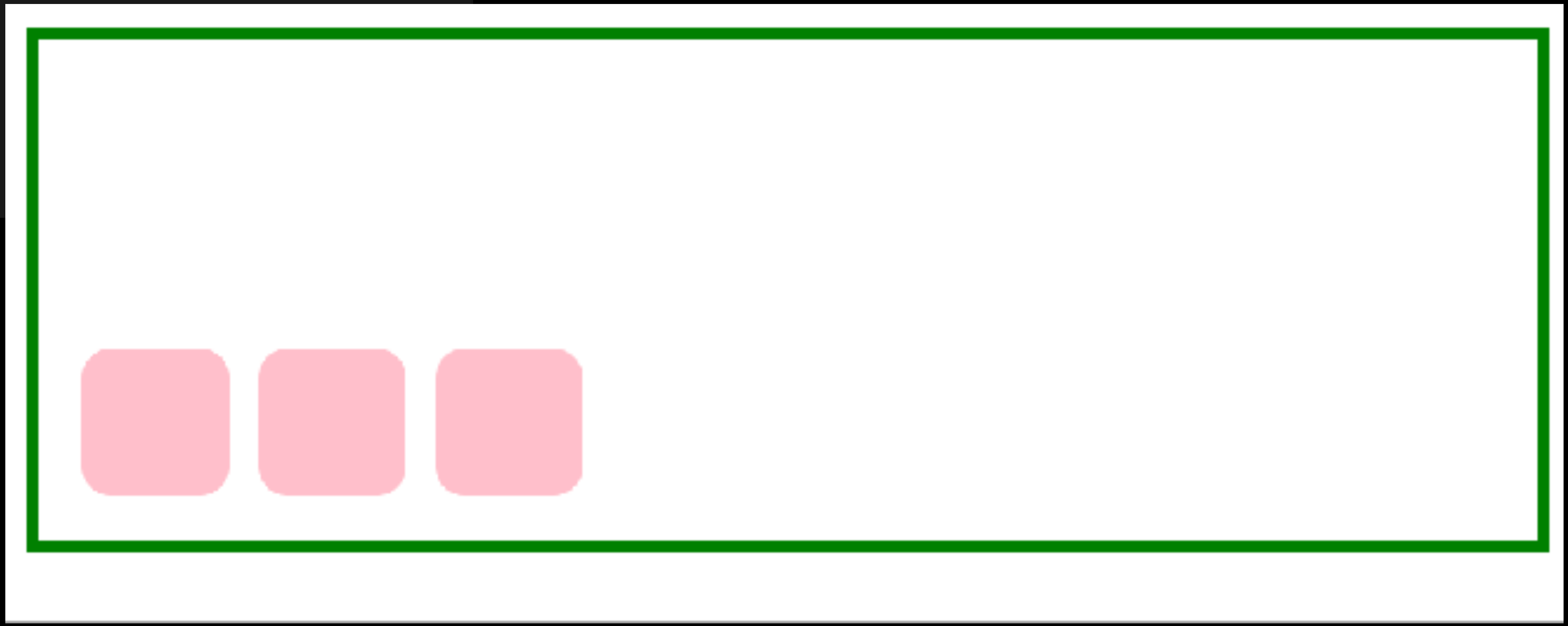


Flex Basics: align-items

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

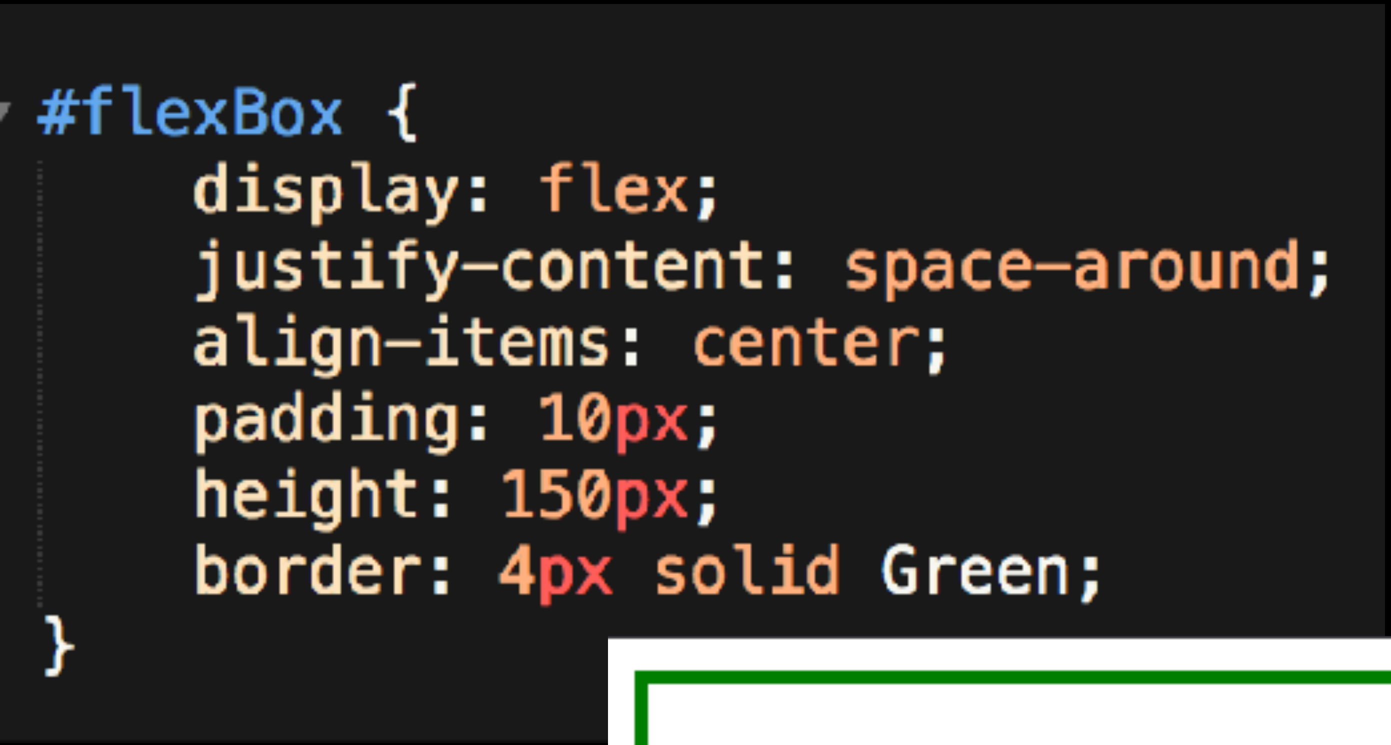


```
#flexBox {
    display: flex;
    align-items: flex-end;
    padding: 10px;
    height: 150px;
    border: 4px solid
}
```

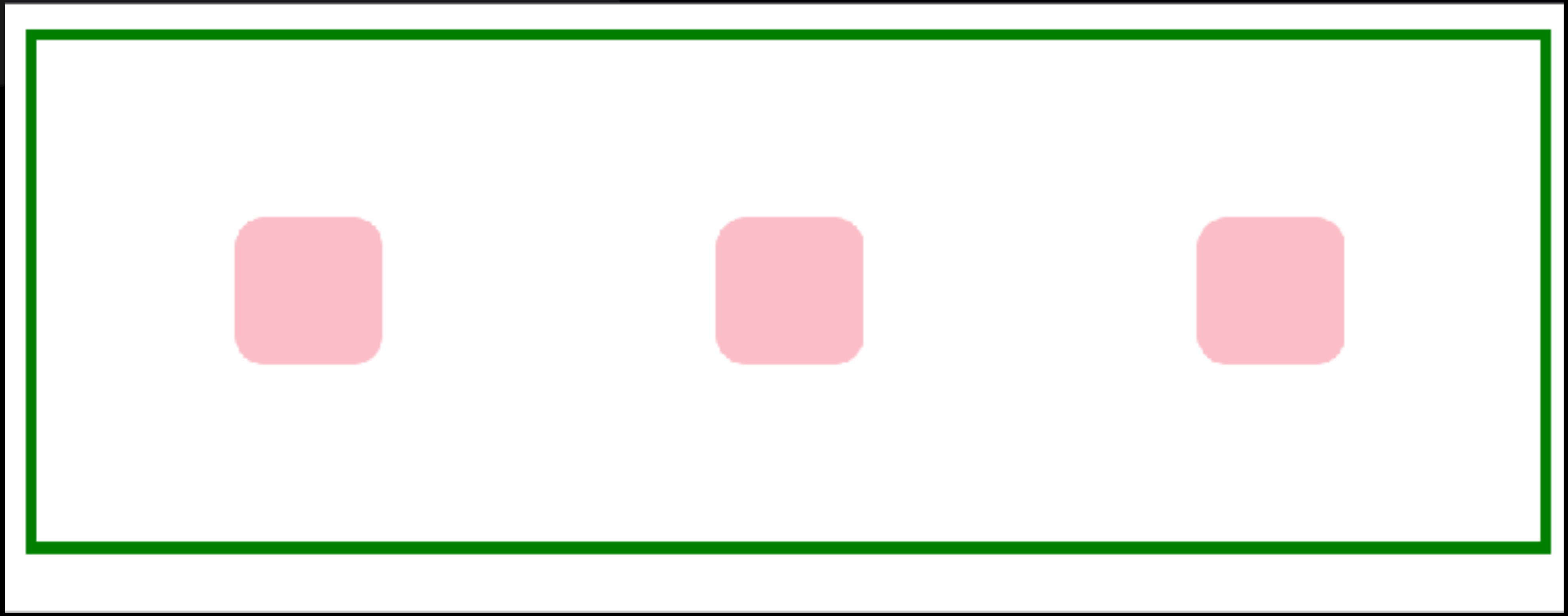


Flex Basics:

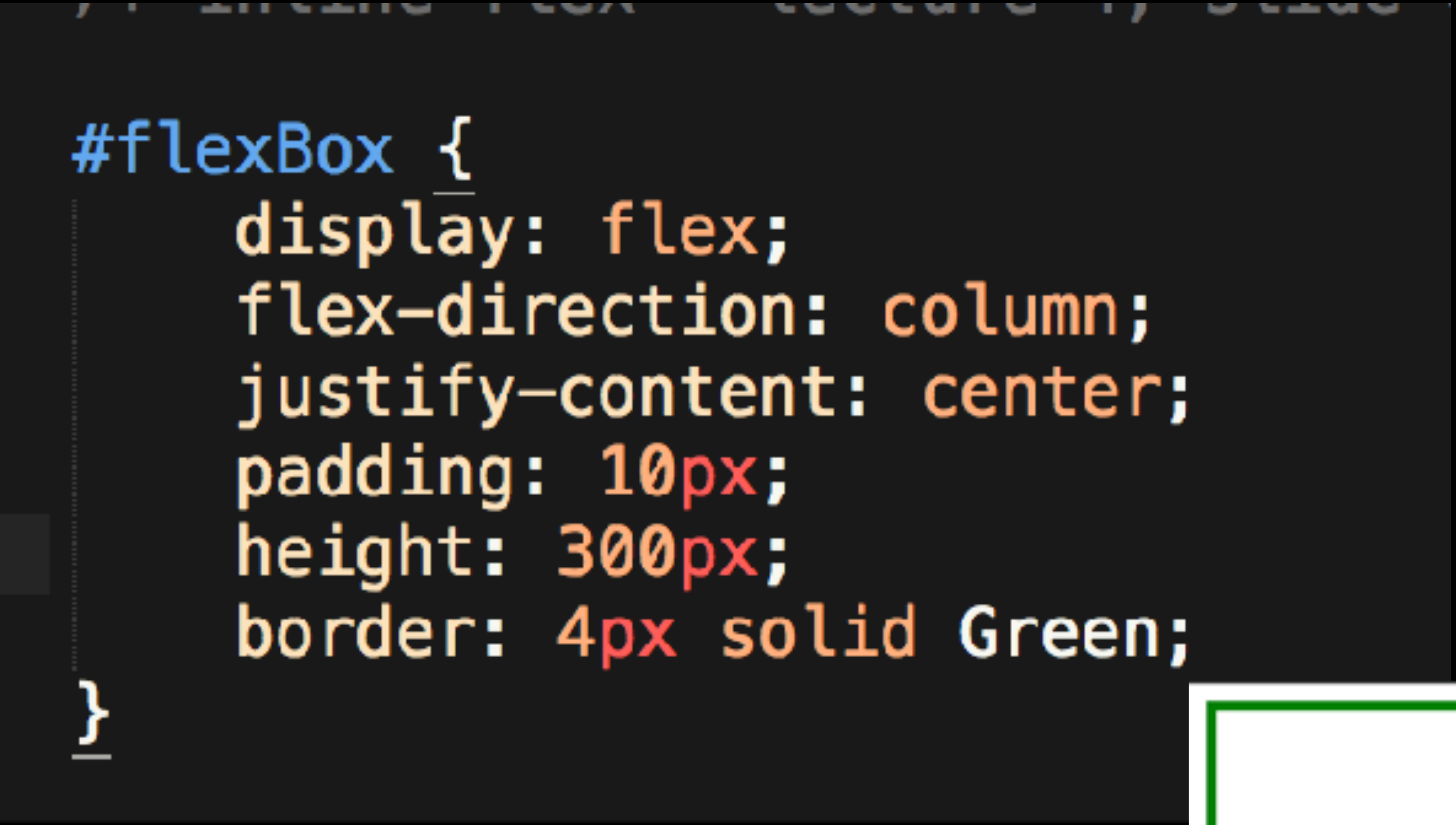
space-between + space-around



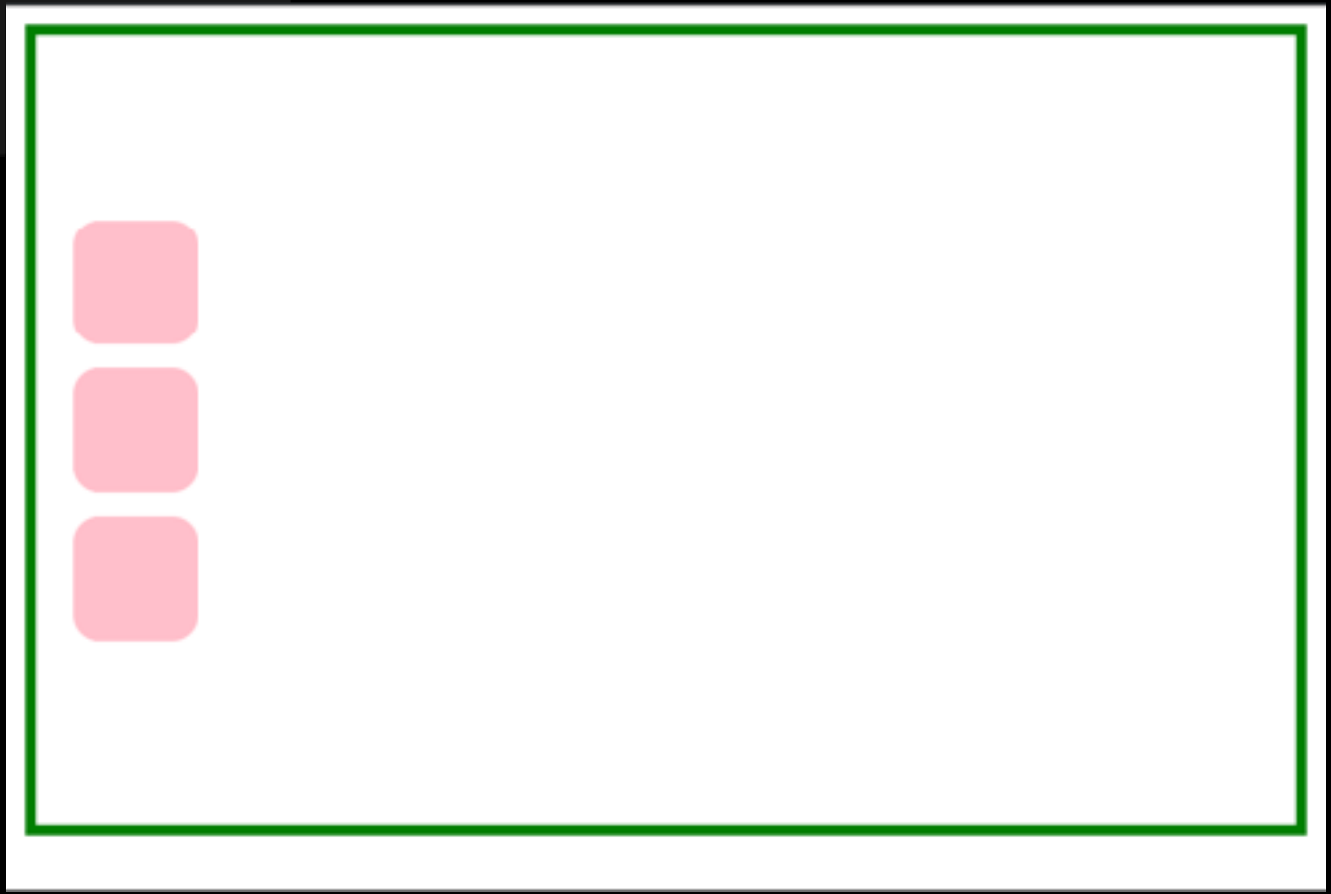
```css
#flexBox {
    display: flex;
    justify-content: space-around;
    align-items: center;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```



Flex Basics: flex-direction



```css
#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 - 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

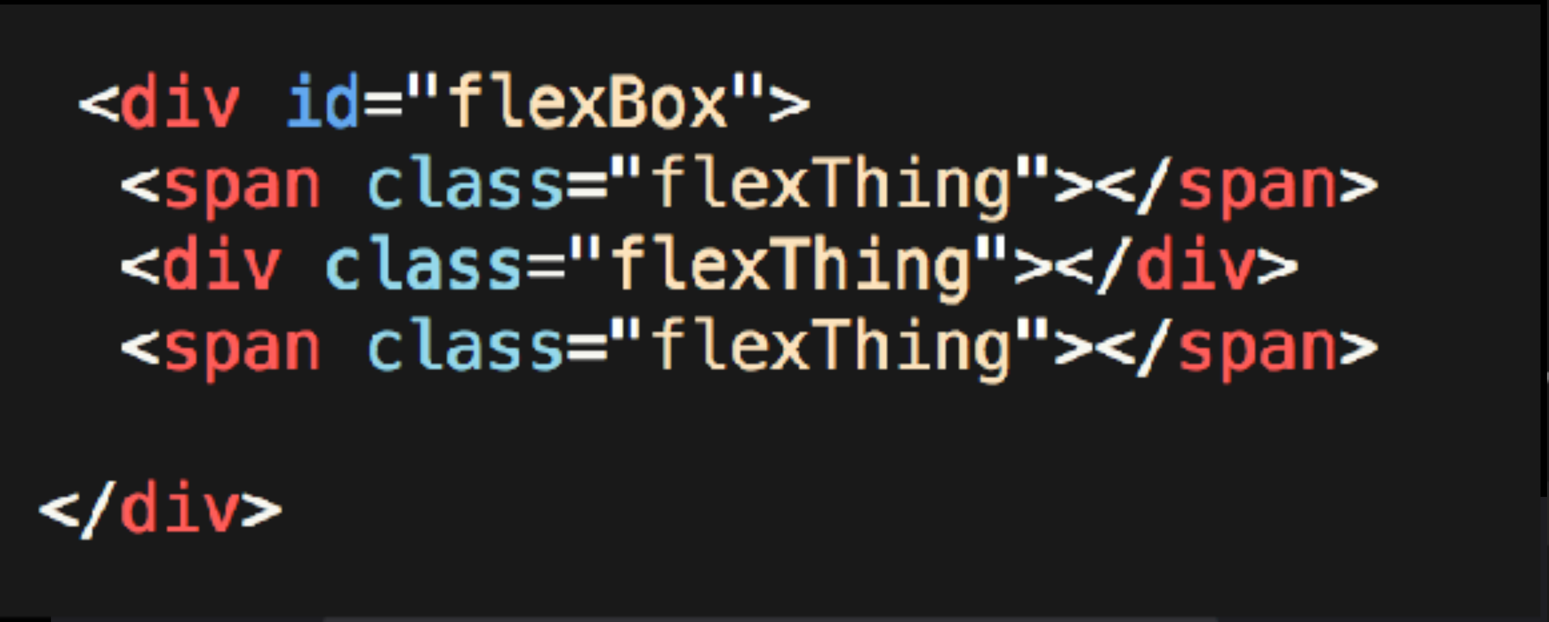
This initial width* of the flex item is called the **flex basis**.

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.

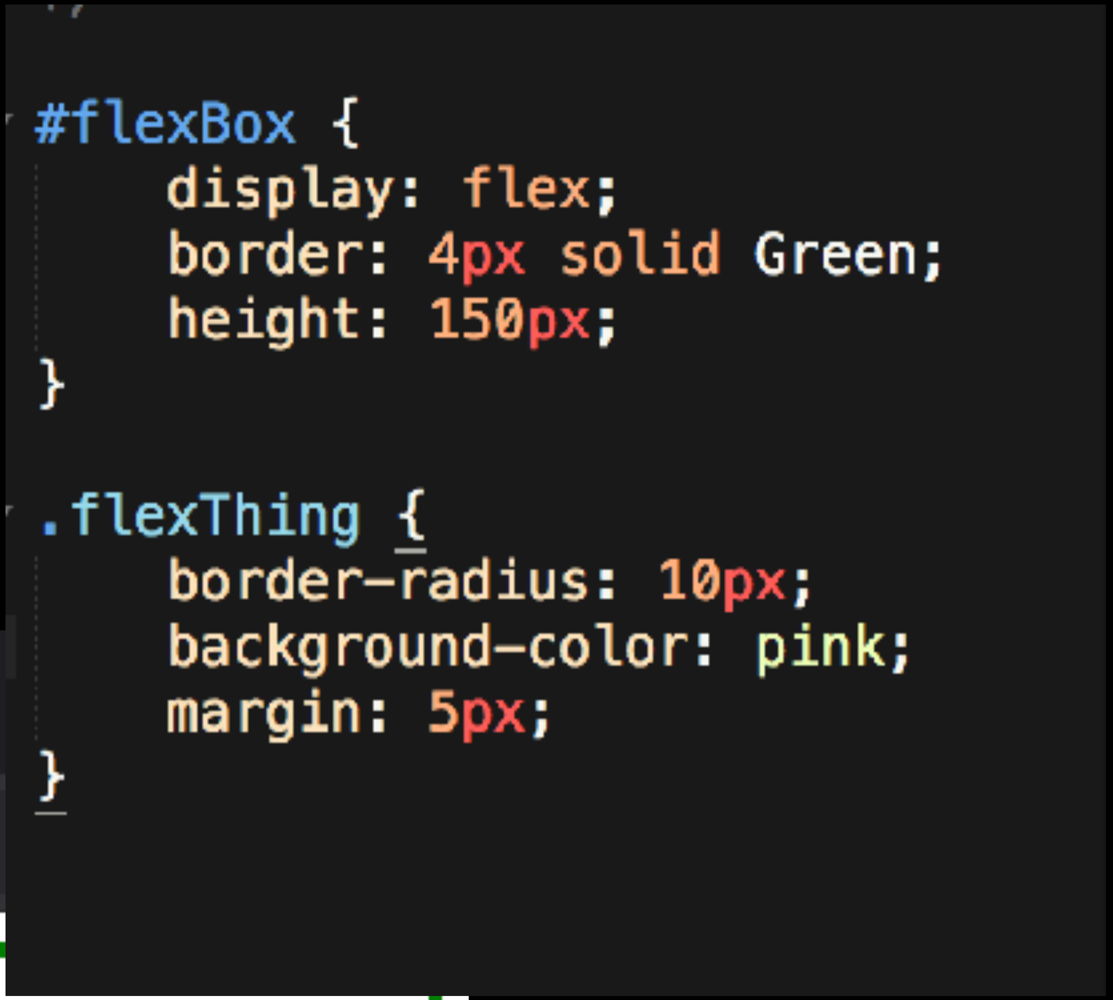
*width in the case of rows; height in the case of columns

Flex Basis

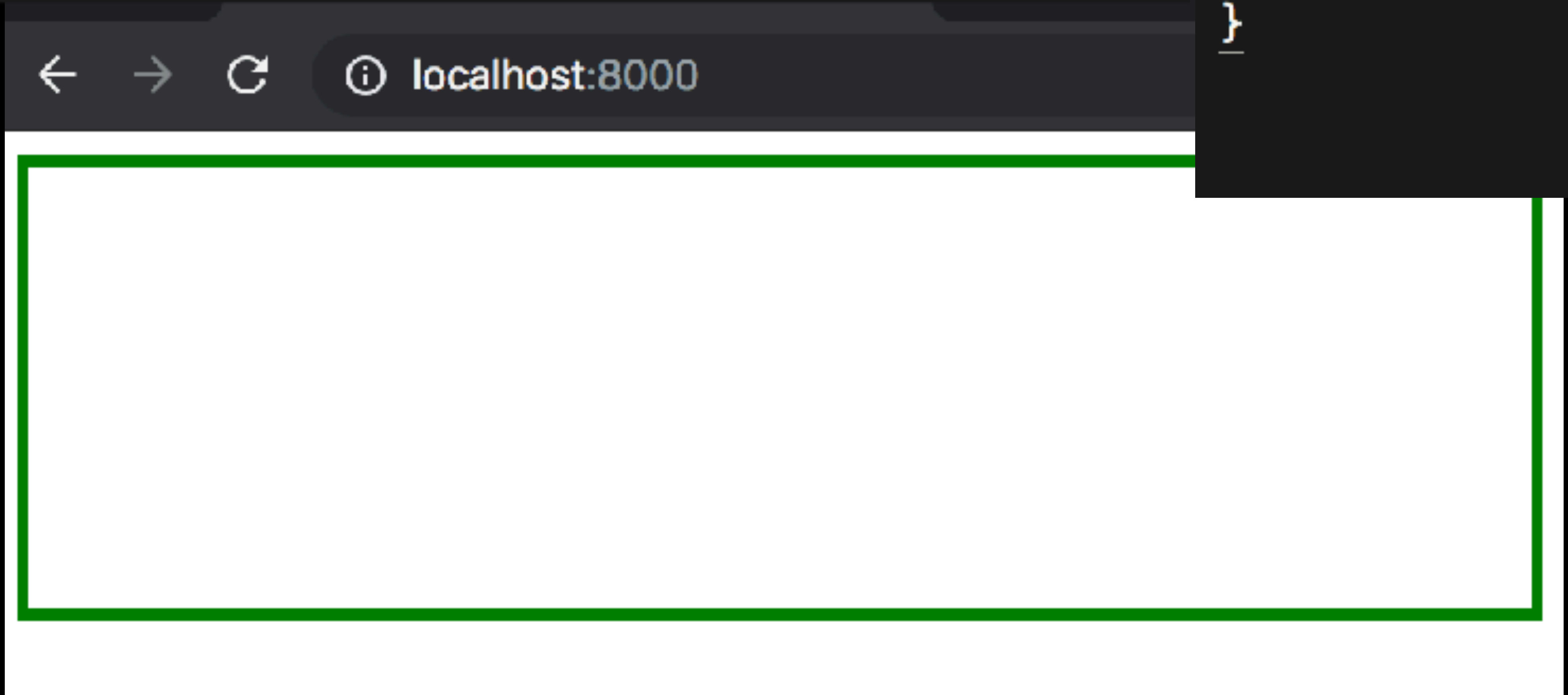
If we unset the height and width, our flex items disappears, because the **flex basis** is now the content size, which is empty:



```
<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;
}
```


localhost:8000