Follow @derekknox

Chapters

+

Light
-

Chapter 6 Deconstructing Designs

U
sing this chapter's title as a clue, what do all the interactive games, tools, and software we love have in common?

Though your answer may be correct, what I'm looking for is that they are all composed of *interconnected rectangles*. This is the case for 2D anyway where 3D products are composed of *interconnected boxes*. The former are typically referred to as *bounding rectangles* and the latter *bounding boxes*. We'll stay focused on 2D.

What this means is that during your various design phases you are—consciously or subconsciously—grouping content into rectangles. Put another way, you are creating a grid. In some grid cells you are creating another grid. And so on.

As you may recall from the *Constructs and Components* section:

> A grid instantly provides the scaffolding, scaling, and patterns of space in which imagery and typography will live.

It is no secret that a grid is extremely useful in visual design. It is even more valuable when coding. In fact, the grid is the shared foundation of the deconstruct and reconstruct processes. We will use this to our advantage and organize this chapter into four parts:

1. Deconstruct
2. Reconstruct Structure
3. Reconstruct Style
4. Reconstruct Behavior

We need a design to apply this process to. This is where Twitter comes in.

#

Twitter

Twitter, and specifically its Tweet Box UI, is a great candidate for deconstructing and reconstructing. This is the case as it bridges the gap between simple and complex.


Tweet Box UI

Below is the zoomed in Tweet Box UI. When viewing it, think through the following before advancing:

- How many rectangles is it comprised of?
- What is their nested structure?
- Where are their boundaries?


Tweet Box UI Zoomed

To reconstruct, we will use our 80/20 learnings from earlier in this book. We will not use the actual HTML, CSS, or JavaScript that Twitter uses. This approach is intentional and will reinforce the power of 80/20 code subsets. Naturally this will help us be productive while achieving more with less.

Even though we focus on deconstructing and reconstructing Twitter's Tweet Box UI in the rest of this chapter, the *exact same process works for your designs*.

#

Process

Before advancing, it is worth previewing the specific process we'll use to deconstruct and reconstruct Twitter's Tweet UI. There are four overarching steps where each has a set of substeps. Each step and substep focuses on a *specific approach to solving a certain problem*. In aggregate, the entire process is used to convert a static design into a dynamic and interactive one.

There is no need to memorize the below steps and substeps. Their presence is solely to reveal a *repeatable and sequential process for converting a static design into a dynamic and interactive one*. Here is the full process:

1. Deconstruct - *Determine the grid and its cells*
    - Rectangle - *Determine the rectangle count*
    - Nest - *Determine the parent-child relationships*
    - Tree - *Create a tree structure using these facts*
2. Reconstruct Structure - *Recreate the structure in HTML*
    - Depth - *Determine each node's nesting depth*
    - Children - *Determine the child count at each node*
    - Grid - *Create the structural grid in code using these facts*
        - Bottom - *Start from the bottom*
        - Depth - *Start at the deepest depth*
        - Wrap - *Update each closing tag's position to wrap its children*
        - Repeat - *Finish each depth before repeating*
3. Reconstruct Style - *Recreate the style with CSS*
    - Placeholder - *Setup placeholder styles to help visualize our grid*
    - Layout - *Setup layout styles*
        - Children - *Child rectangle focus*
        - Space - *Empty space focus*
    - Content - *Setup content styles*
4. Reconstruct Behavior - *Recreate the behavior with JavaScript*
    - Change - *Determine what changes at execution time*
    - Depend - *Determine the change dependencies*
    - Component - *Create components using these facts*
        - Shell - *Code the component shell*
        - Interface - *Code the component interface*
        - Implementation - *Code the component implementation*

#

Deconstruct

Though there are always variations in rectangle count for a given deconstructed solution, ours will be composed of twenty-seven. Was your answer closer to eleven? If so, there is one fundamental area of improvement for you to focus on. Remember, we are designing for *dynamic* not *static* layouts. Tactically this translates to thinking more in:

- Percentages and ratios vs. exact pixel dimensions
- Content anchored in nested containers vs. absolutely positioned within a single container

> Thinking and designing for dynamic—not static—layouts is the single biggest step to leveling up as a designer.

Twenty-seven may seem like a high number for such a simple UI, but when thinking and designing for dynamic layouts this isn't surprising. How exactly did we come to this twenty-seven number though? There are two core approaches for tracing out the rectangles:

1. Mentally
2. In your design tool of choice

You will improve with the former approach in time. If you consistently nest and organize the layers in your design tool of choice, you are well set for the second approach. In either case, the deconstruction process has three core substeps:

1. Rectangle - *Determine the rectangle count*
2. Nest - *Determine the parent-child relationships*
3. Tree - *Create a tree structure using these facts*

Each nesting relates to the parent-child relationship we talked about earlier in the *Structure* section. Below is the result of our three step deconstruction process using an HTML-like code. Each tree node is composed of:

- An implied rectangle
- A proposed HTML element
- Contextual information

```
<div> main
    <div> avatar
        <img> avatar graphic
    <div> actions
        <div> top
            <p> What's happening? text
        <div> bottom
            <div> left
                <button> photo
                    <img> photo graphic
                <button> gif
                    <img> gif graphic
                <button> poll
                    <img> poll graphic
                <button> emoji
                    <img> emoji graphic
                <button> more...
                    <img> more... graphic
            <div> right
                <div> progress bar
                    <img> progress bar graphic
                <div> divider
                    <img> divider graphic
                <button> add
                    <img> plus graphic
                <button> Tweet
                    <span> Tweet text
```

#

Reconstruct Structure

Bonus points to you if you noticed the tree is nested six levels deep. This information is extremely valuable as it becomes our starting point for the reconstruction process. This process consists of three core substeps:

1. Depth - *Determine each node's nesting depth*
2. Children - *Determine the child count at each node*
3. Grid - *Create the structural grid in code using these facts*

#

Depth

Building off our tree from the deconstruct process, we can update it with the depth information:

```
<div> main (depth 1)
    <div> avatar (depth 2)
        <img> avatar graphic (depth 3)
    <div> actions (depth 2)
        <div> top (depth 3)
            <p> What's happening? text (depth 4)
        <div> bottom (depth 3)
            <div> left (depth 4)
                <button> photo (depth 5)
                    <img> photo graphic (depth 6)
                <button> gif (depth 5)
                    <img> gif graphic (depth 6)
                <button> poll (depth 5)
                    <img> poll graphic (depth 6)
                <button> emoji (depth 5)
                    <img> emoji graphic (depth 6)
                <button> more... (depth 5)
                    <img> more... graphic (depth 6)
            <div> right (depth 4)
                <div> progress bar (depth 5)
                    <img> progress bar graphic (depth 6)
                <div> divider (depth 5)
                    <img> divider graphic (depth 6)
                <button> add (depth 5)
                    <img> plus graphic (depth 6)
                <button> Tweet (depth 5)
                    <span> Tweet text (depth 6)
```

#

Children

We can further update our tree with the child count information. Take note that we only count *immediate* children.

```
<div> main (depth 1) (children 2)
    <div> avatar (depth 2) (children 1)
        <img> avatar graphic (depth 3) (children 0)
    <div> actions (depth 2) (children 2)
        <div> top (depth 3) (children 1)
            <p> What's happening? text (depth 4) (children 0)
        <div> bottom (depth 3) (children 2)
            <div> left (depth 4) (children 5)
                <button> photo (depth 5) (children 1)
                    <img> photo graphic (depth 6) (children 0)
                <button> gif (depth 5) (children 1)
                    <img> gif graphic (depth 6) (children 0)
                <button> poll (depth 5) (children 1)
                    <img> poll graphic (depth 6) (children 0)
                <button> emoji (depth 5) (children 1)
                    <img> emoji graphic (depth 6) (children 0)
                <button> more... (depth 5) (children 1)
                    <img> more... graphic (depth 6) (children 0)
            <div> right (depth 4) (children 4)
                <div> progress bar (depth 5) (children 1)
                    <img> progress bar graphic (depth 6) (children 0)
                <div> divider (depth 5) (children 1)
                    <img> divider graphic (depth 6) (children 0)
                <button> add (depth 5) (children 1)
                    <img> plus graphic (depth 6) (children 0)
                <button> Tweet (depth 5) (children 1)
                    <span> Tweet text (depth 6) (children 0)
```

#

Grid

Now we have all the information we need to recreate our grid in code. In fact, because our tree has a proposed HTML element, we can simply close each element's tag. This will give us the structure we're looking for. Sweet! Now our tree looks like:

```
<div> main (depth 1) (children 2) </div>
    <div> avatar (depth 2) (children 1) </div>
        <img> avatar graphic (depth 3) (children 0) </img>
    <div> actions (depth 2) (children 2) </div>
```

```
        <div> top (depth 3) (children 1) </div>
            <p> What's happening? text (depth 4) (children 0) </p>
        <div> bottom (depth 3) (children 2) </div>
            <div> left (depth 4) (children 5) </div>
                <button> photo (depth 5) (children 1) </button>
                    <img> photo graphic (depth 6) (children 0) </img>
                <button> gif (depth 5) (children 1) </button>
                    <img> gif graphic (depth 6) (children 0) </img>
                <button> poll (depth 5) (children 1) </button>
                    <img> poll graphic (depth 6) (children 0) </img>
                <button> emoji (depth 5) (children 1) </button>
                    <img> emoji graphic (depth 6) (children 0) </img>
                <button> more... (depth 5) (children 1) </button>
                    <img> more... graphic (depth 6) (children 0) </img>
            <div> right (depth 4) (children 4) </div>
                <div> progress bar (depth 5) (children 1) </div>
                    <img> progress bar graphic (depth 6) (children 0) </img>
                <div> divider (depth 5) (children 1) </div>
                    <img> divider graphic (depth 6) (children 0) </img>
                <button> add (depth 5) (children 1) </button>
                    <img> plus graphic (depth 6) (children 0) </img>
                <button> Tweet (depth 5) (children 1) </button>
                    <span> Tweet text (depth 6) (children 0) </span>
```

As you may have thought when reading the above update, this won't actually work. The reason is that our closing HTML element tags *do not wrap their children*. In order to wrap them properly, we need to use our depth and child count information. With this information we can follow the below sequence:

1. Bottom - *Start from the bottom*
2. Depth - *Start at the deepest depth*
3. Wrap - *Update each closing tag's position to wrap its children*
4. Repeat - *Finish each depth before repeating*

For example the:

```
<button> Tweet (depth 5) (children 1) </button>
    <span> Tweet text (depth 6) (children 0) </span>
```

becomes:

```
<button> Tweet (depth 5) (children 1)
    <span> Tweet text (depth 6) (children 0) </span>
</button>
```

During this process you simply need to pay attention to the child count of each node. If it is `0` then you can skip it. Otherwise you need to update the closing tag's position so that it wraps its child nodes. Simple as that. When complete the result is:

```
<div> main (depth 1) (children 2)
    <div> avatar (depth 2) (children 1)
        <img> avatar graphic (depth 3) (children 0) </img>
    </div>
    <div> actions (depth 2) (children 2)
        <div> top (depth 3) (children 1)
            <p> What's happening? text (depth 4) (children 0) </p>
        </div>
        <div> bottom (depth 3) (children 2)
            <div> left (depth 4) (children 5)
                <button> photo (depth 5) (children 1)
                    <img> photo graphic (depth 6) (children 0) </img>
                </button>
                <button> gif (depth 5) (children 1)
                    <img> gif graphic (depth 6) (children 0) </img>
                </button>
                <button> poll (depth 5) (children 1)
                    <img> poll graphic (depth 6) (children 0) </img>
                </button>
                <button> emoji (depth 5) (children 1)
                    <img> emoji graphic (depth 6) (children 0) </img>
                </button>
                <button> more... (depth 5) (children 1)
                    <img> more... graphic (depth 6) (children 0) </img>
                </button>
            </div>
```

```
            <div> right (depth 4) (children 4)
                <div> progress bar (depth 5) (children 1)
                    <img> progress bar graphic (depth 6) (children 0) </img>
                </div>
                <div> divider (depth 5) (children 1)
                    <img> divider graphic (depth 6) (children 0) </img>
                </div>
                <button> add (depth 5) (children 1)
                    <img> plus graphic (depth 6) (children 0) </img>
                </button>
                <button> Tweet (depth 5) (children 1)
                    <span> Tweet text (depth 6) (children 0) </span>
                </button>
            </div>
        </div>
    </div>
</div>
```

If you were to view this structured HTML in a browser, the rendered output would admittedly look ugly and seemingly disorganized. In fact, it would look like this:

Structure with Depth, Children, and Grid

This looks horrible visually, but it is structurally correct. We will fix this aesthetic issue now by applying style.

Before advancing, take note of the "Open Sandbox" button in the example above. All remaining examples in this chapter have it too. Click or tap it to learn, explore, and edit the code yourself!

#

Reconstruct Style

This process consists of three core substeps:

1. Placeholder - *Setup placeholder styles to help visualize our grid*
2. Layout - *Setup layout styles*
    - Children - *Child rectangle focus*
    - Space - *Empty space focus*
3. Content - *Setup content styles*

#

Placeholder

Since we already did the work of identifying the depth of each node, we can simply convert our various "(depth X)" notes into `class`es. For example:

```
<div> main (depth 1) (children 2)
```

becomes:

```
<div class='depth-1'> main (children 2)
```

Before manually updating each node, I highly recommend looking into a code editor that supports *multiple cursors*. Welcome to a superpower. It took only *twenty seconds* to update the HTML tree with the `class` definition at each node using multiple cursors. Coders like shortcuts remember? As you may recall from the *Style* section:

> A professional's level of experience simply makes him or her quicker in their application of it. Professional coders—just like beginners—still need to reference resources.

We refresh on this for two reasons:

1. Experience manifests as tools knowledge, not just speed
2. Style resource referencing was needed to complete this section and this book!

The multiple cursors feature is perfect for facilitating this `class='depth-x'` update efficiently, but the manual approach works well too. Regardless of the approach, the full update looks like this:

```
<div class='depth-1'> main (children 2)
    <div class='depth-2'> avatar (children 1)
        <img class='depth-3'> avatar graphic (children 0) </img>
    </div>
    <div class='depth-2'> actions (children 2)
        <div class='depth-3'> top (children 1)
            <p class='depth-4'> What's happening? text (children 0) </p>
        </div>
        <div class='depth-3'> bottom (children 2)
            <div class='depth-4'> left (children 5)
                <button class='depth-5'> photo (children 1)
                    <img class='depth-6'> photo graphic (children 0) </img>
                </button>
                <button class='depth-5'> gif (children 1)
                    <img class='depth-6'> gif graphic (children 0) </img>
                </button>
                <button class='depth-5'> poll (children 1)
                    <img class='depth-6'> poll graphic (children 0) </img>
                </button>
                <button class='depth-5'> emoji (children 1)
                    <img class='depth-6'> emoji graphic (children 0) </img>
                </button>
                <button class='depth-5'> more... (children 1)
                    <img class='depth-6'> more... graphic (children 0) </img>
                </button>
            </div>
            <div class='depth-4'> right (children 4)
                <div class='depth-5'> progress bar (children 1)
                    <img class='depth-6'> progress bar graphic (children 0) </img>
                </div>
                <div class='depth-5'> divider (children 1)
                    <img class='depth-6'> divider graphic (children 0) </img>
                </div>
                <button class='depth-5'> add (children 1)
                    <img class='depth-6'> plus graphic (children 0) </img>
                </button>
                <button class='depth-5'> Tweet (children 1)
                    <span class='depth-6'> Tweet text (children 0) </span>
                </button>
            </div>
        </div>
    </div>
</div>
```

After updating all our HTML element nodes with the proper `class` placeholders, we simply need to update our CSS so that we actually define each of these placeholder styles. The color choice for each placeholder is up to you. The only goal influencing color choice is to ensure they are clearly differentiated. Since we have six depths and there are seven distinct colors in the rainbow, we'll start with those:

```
.depth-1 { background-color: red;     /* #FF0000 */ }
.depth-2 { background-color: orange;  /* #FFA500 */ }
.depth-3 { background-color: yellow;  /* #FFFF00 */ }
.depth-4 { background-color: green;   /* #00FF00 */ }
.depth-5 { background-color: blue;    /* #0000FF */ }
.depth-6 { background-color: indigo;  /* #4B0082 */ }
```

Now that each depth container is clearly differentiated by color, we can now more easily see our style updates in the browser as we begin to iterate toward our layout and content CSS styles.

#

Layout - Children

Now we want to fix our layout. Put another way, we need to properly *distribute the children and space within the parent rectangles*. The `class` attribute enables this.

We'll focus on the child rectangles to start, so we'll use either `layout-horizontal` or `layout-vertical`. An update is needed only if the child count is two or more. For example:

```
<div class='depth-1'> main (children 2)
```

updates to:

```
<div class='depth-1 layout-horizontal'> main (children 2)
```

but:

```
<div class='depth-2'> avatar (children 1)
```

remains the same. When we update our entire tree we get:

```
<div class='depth-1 layout-horizontal'> main (children 2)
    <div class='depth-2'> avatar (children 1)
        <img class='depth-3'> avatar graphic (children 0) </img>
    </div>
    <div class='depth-2 layout-vertical'> actions (children 2)
        <div class='depth-3 layout-horizontal'> top (children 1)
            <p class='depth-4'> What's happening? text (children 0) </p>
        </div>
        <div class='depth-3 layout-horizontal'> bottom (children 2)
            <div class='depth-4 layout-horizontal'> left (children 5)
                <button class='depth-5'> photo (children 1)
                    <img class='depth-6'> photo graphic (children 0) </img>
                </button>
                <button class='depth-5'> gif (children 1)
                    <img class='depth-6'> gif graphic (children 0) </img>
                </button>
                <button class='depth-5'> poll (children 1)
                    <img class='depth-6'> poll graphic (children 0) </img>
                </button>
                <button class='depth-5'> emoji (children 1)
                    <img class='depth-6'> emoji graphic (children 0) </img>
                </button>
                <button class='depth-5'> more... (children 1)
                    <img class='depth-6'> more... graphic (children 0) </img>
                </button>
            </div>
            <div class='depth-4 layout-horizontal'> right (children 4)
                <div class='depth-5'> progress bar (children 1)
                    <img class='depth-6'> progress bar graphic (children 0) </img>
                </div>
                <div class='depth-5'> divider (children 1)
                    <img class='depth-6'> divider graphic (children 0) </img>
                </div>
                <button class='depth-5'> add (children 1)
                    <img class='depth-6'> plus graphic (children 0) </img>
                </button>
                <button class='depth-5'> Tweet (children 1)
                    <span class='depth-6'> Tweet text (children 0) </span>
                </button>
            </div>
        </div>
    </div>
</div>
```

After this initial layout pass our rendered structure begins to look closer to our goal. It is still quite a ways off however.

Style with Layout Children Placeholders

Our various `depth-X` classes combined with our added `layout-horizontal` and `layout-vertical` class definitions enable the above render. The class definitions are:

```
.layout-horizontal {
    display: flex;
    flex-direction: row;
}

.layout-vertical {
    display: flex;
    flex-direction: column;
}
```

Since we don't need our children count information anymore, we can further cleanup our layout styling. To do so we'll simply remove all our notes that helped us up to this point. If our end design uses specific text then we'll leave that text. For example:

```
<p class='depth-4'> What's happening? text (children 0) </p>
```

becomes:

```
<p class='depth-4'>What's happening?</p>
```

and:

```
<span class='depth-6'> Tweet text (children 0) </span>
```

becomes:

```
<span class='depth-6'>Tweet</span>
```

Take note that we also cleanup the spacing between each element's opening and closing tags. Our end result becomes:

```
<div class='depth-1 layout-horizontal'>
    <div class='depth-2'>
        <img class='depth-3'></img>
    </div>
    <div class='depth-2 layout-vertical'>
        <div class='depth-3 layout-horizontal'>
            <p class='depth-4'>What's happening?</p>
        </div>
        <div class='depth-3 layout-horizontal'>
            <div class='depth-4 layout-horizontal'>
                <button class='depth-5'>
                    <img class='depth-6'></img>
                </button>
                <button class='depth-5'>
                    <img class='depth-6'></img>
                </button>
                <button class='depth-5'>
                    <img class='depth-6'></img>
                </button>
                <button class='depth-5'>
                    <img class='depth-6'></img>
                </button>
                <button class='depth-5'>
                    <img class='depth-6'></img>
                </button>
            </div>
            <div class='depth-4 layout-horizontal'>
                <div class='depth-5'>
                    <img class='depth-6'></img>
                </div>
                <div class='depth-5'>
                    <img class='depth-6'></img>
                </div>
                <button class='depth-5'>
                    <img class='depth-6'></img>
                </button>
                <button class='depth-5'>
                    <span class='depth-6'>Tweet</span>
                </button>
            </div>
        </div>
```

```
    </div>
</div>
```

When viewing this tree rendered in the browser, it may seem like we took a step backward. It appears this way because we have yet to define *empty space styles* for our containers.

#

Layout - Space

We'll start by adding a placeholder style addition of `padding: 5px; margin: 5px;` to each `depth-X` class definition. When rendered now, we see that we are still on the right track.

Style with Layout Space Placeholders

Let's leave this `padding: 5px; margin: 5px;` placeholder as it improves visualizing our progress as we iterate.

To finish off our layout step we need a few of our containers to properly handle *empty space*. Similar to how our `layout-horizontal` and `layout-vertical` styles help layout *child rectangles*, we also need styles to layout *empty space*. The first will be used to tell a container how to *grow into* empty space and the second and third will tell a container how to *distribute* empty space. The class names we'll use are:

- `empty-space-grow`
- `empty-space-between`
- `empty-space-around`

Based off these class names and the Style with Layout Space Placeholders render from above, try to guess which containers need them. Here is a clue:

- `empty-space-grow` (two instances)
- `empty-space-between` (one instance)
- `empty-space-around` (nine instances)

Combined, they result in this render:

Style with Layout Space

There are two core changes needed to achieve this result. The first being:

```
<!-- The above code removed for brevity -->
<div class='depth-2 layout-vertical empty-space-grow'>
    <div class='depth-3 layout-horizontal'>
        <p class='depth-4 empty-space-grow'>What's happening?</p>
    </div>
    <div class='depth-3 layout-horizontal empty-space-between'>
<!-- The below code removed for brevity -->
```

The second core change will also include a Layout - Children style addition. We learned previously that an update is needed, "...only if the child count is two or more." This is still the correct default rule to follow. As seen in the rendered

output however, it becomes obvious that an iterative update is required to vertically center some of the content. Naturally, the progress bar, divider, and all `<button>` elements get this update. The result becomes:

```
<div class='depth-1 layout-horizontal'>
    <div class='depth-2'>
        <img class='depth-3'></img>
    </div>
    <div class='depth-2 layout-vertical empty-space-grow'>
        <div class='depth-3 layout-horizontal'>
            <p class='depth-4 empty-space-grow'>What's happening?</p>
        </div>
        <div class='depth-3 layout-horizontal empty-space-between'>
            <div class='depth-4 layout-horizontal'>
                <button class='depth-5 layout-vertical empty-space-around'>
                    <img class='depth-6'></img>
                </button>
                <button class='depth-5 layout-vertical empty-space-around'>
                    <img class='depth-6'></img>
                </button>
                <button class='depth-5 layout-vertical empty-space-around'>
                    <img class='depth-6'></img>
                </button>
                <button class='depth-5 layout-vertical empty-space-around'>
                    <img class='depth-6'></img>
                </button>
                <button class='depth-5 layout-vertical empty-space-around'>
                    <img class='depth-6'></img>
                </button>
            </div>
            <div class='depth-4 layout-horizontal'>
                <div class='depth-5 layout-vertical empty-space-around'>
                    <img class='depth-6'></img>
                </div>
                <div class='depth-5 layout-vertical empty-space-around'>
                    <img class='depth-6'></img>
                </div>
                <button class='depth-5 layout-vertical empty-space-around'>
                    <img class='depth-6'></img>
                </button>
                <button class='depth-5 layout-vertical empty-space-around'>
                    <span class='depth-6'>Tweet</span>
                </button>
            </div>
        </div>
    </div>
</div>
```

The `class` definitions for these additions are:

```
.empty-space-around {
    justify-content: space-around;
}

.empty-space-between {
    justify-content: space-between;
}

.empty-space-grow {
    flex-grow: 1;
}
```

The step order in this section is intentional and gets you at least an 80/20 result. It's important to remember that the closer we get to the content step, the more we'll have to tweak our initial structure and style to manifest our target design.

Now that we are finished with our Layout - Space step, we can finally move on to design *content*.

#

Content

With our placeholder and layout styling in place, we now have a great baseline to iterate on. In comparing our current render with our target design, the next major differences relate to imagery. Or to be more precise, the lack of imagery. As you may recall from the *Constructs and Components* section there are three categories of imagery:

- Photography
- Iconography
- Illustration

We can organize our updates using these categories:

- Photography
  - Avatar
- Iconography
  - Photo
  - Gif
  - Poll
  - Emoji
  - More...
  - Divider
  - Add
- Illustration
  - Progress Bar

Put another way, we simply need to update all the `<img>` element tags with their correct `src` attribute and value. As designers we know that `.jpeg`, `.png`, and `.gif` are common raster formats (useful in photography) where `.svg` is the most common vector format (useful in iconography). Illustrations often straddle this line and since we have to update the Progress Bar at execution time, we'll use a placeholder `.svg` version until the final Reconstruct Behavior step.

As we learned how to link to asset files in *Anatomy of HTML, CSS, and JavaScript* we simply update each `src` attribute to point to the desired image file. We end up with the following render:

Style Content with Imagery and Placeholders

Now to improve our content style, we'll add two `class`es to enforce the desired `width` and `height` dimensions of some content. The numbers below are not made up. They were confirmed on Twitter's website via the browser's developer tools that were first mentioned in the *Debugger Statement* section. You could easily create a bounding box rectangle in your design tool of choice to come to the same conclusions.

In both Twitter's code and your design tool of choice, the units are in *pixels*. We will instead use `rem` (*root element font-size*) units. Why? As you may recall from the Deconstruct section:

> Thinking and designing for dynamic—not static—layouts is the single biggest step to leveling up as a designer.

This means we need to think in *percentages and ratios vs. exact pixel dimensions*. The `rem` unit enables this. There are online conversion tools to help you convert from `px` to `rem` and vice-versa. A common base `font-size: 16px` is `1rem` for example and the one we use. Using `rem`, our added styles are:

```
.graphic-avatar {
    border-radius: 50%;
    height: 3rem;
    width: 3rem;
}
```

We'll additionally add some button related classes to ensure we override the *browser default styles* first mentioned in the *Style* section. We update each `<button>` with the `button` class, selectively apply the `button-tweet-media` class, and apply the `button-tweet` to the Tweet button:

```
.button {
    background-color: transparent;
    border: 0;
    cursor: pointer;
```

```
    padding: 0;
}

.button-tweet-media {
    margin: 0.25rem 0.4rem;
}

.button-tweet {
    background-color: #1DA1F2;
    border-radius: 2rem;
    color: #FFFFFF;
    height: 2.5rem;
    margin-left: 0.5rem;
}
```

With the above content style updates we get the following render:

Style Content with Placeholders

If we temporarily remove our placeholder styles we get:

Style Content without Placeholders

We're so close! We only have minor content spacing styles left to apply. Since this is our focus, we no longer need our `padding: 5px; margin: 5px;` helpers. Once removed, our render becomes:

Style Content with Color Placeholders Only

We can see that there are a few target areas in need of content spacing:

- Tweet Box
- "What's happening?" text
- Tweet text

The following `class`es get us 99% of what we want style-wise:

```
.content-tweet-box {
    padding: 0.5rem 1rem;
}

.content-tweet-text {
    color: grey;
```

```
    font-size: 1.4rem;
    font-weight: 300;
    margin: 0.75rem 0.5rem 2rem 1rem;
}

.content-tweet-label {
    font-size: 1rem;
    font-weight: 600;
    margin: 0 1rem 0 1rem;
}
```

When we completely remove our placeholders our structure becomes:

```
<div class='layout-horizontal content-tweet-box'>
    <div>
        <img class='graphic-avatar' src='assets/img/avatar.png'></img>
    </div>
    <div class='layout-vertical empty-space-grow'>
        <div class='layout-horizontal'>
            <p class='empty-space-grow content-tweet-text'>What's happening?</p>
        </div>
        <div class='layout-horizontal empty-space-between'>
            <div class='layout-horizontal'>
                <button class='layout-vertical empty-space-around button button-tweet-media'>
                    <img src='assets/img/photo.svg'></img>
                </button>
                <button class='layout-vertical empty-space-around button button-tweet-media'>
                    <img src='assets/img/gif.svg'></img>
                </button>
                <button class='layout-vertical empty-space-around button button-tweet-media'>
                    <img src='assets/img/poll.svg'></img>
                </button>
                <button class='layout-vertical empty-space-around button button-tweet-media'>
                    <img src='assets/img/emoji.svg'></img>
                </button>
                <button class='layout-vertical empty-space-around button button-tweet-media'>
                    <img src='assets/img/more.svg'></img>
                </button>
            </div>
            <div class='layout-horizontal'>
                <div class='layout-vertical empty-space-around'>
                    <img src='assets/img/progress-bar.svg'></img>
                </div>
                <div class='layout-vertical empty-space-around'>
                    <img src='assets/img/divider.svg'></img>
                </div>
                <button class='layout-vertical empty-space-around button'>
                    <img src='assets/img/add.svg'></img>
                </button>
                <button class='layout-vertical empty-space-around button button-tweet'>
                    <span class='content-tweet-label'>Tweet</span>
                </button>
            </div>
        </div>
    </div>
</div>
```

And our style becomes:

```
/* Layout - Children */

.layout-horizontal {
    display: flex;
    flex-direction: row;
}

.layout-vertical {
    display: flex;
    flex-direction: column;
}

/* Layout - Space */

.empty-space-around {
    justify-content: space-around;
}
```

```
.empty-space-between {
    justify-content: space-between;
}

.empty-space-grow {
    flex-grow: 1;
}

/* Layout - Content */

.graphic-avatar {
    border-radius: 50%;
    height: 3rem;
    width: 3rem;
}

.button {
    background-color: transparent;
    border: 0;
    cursor: pointer;
    padding: 0;
}

.button-tweet-media {
    margin: 0.25rem 0.4rem;
}

.button-tweet {
    background-color: #1DA1F2;
    border-radius: 2rem;
    color: #FFFFFF;
    height: 2.5rem;
    margin-left: 0.5rem;
}

.content-tweet-box {
    padding: 0.5rem 1rem;
}

.content-tweet-text {
    color: grey;
    font-size: 1.4rem;
    font-weight: 300;
    margin: 0.75rem 0.5rem 2rem 1rem;
}

.content-tweet-label {
    font-size: 1rem;
    font-weight: 600;
    margin: 0 1rem 0 1rem;
}
```

Using these structure and style definitions, we get a Tweet Box whose layout is responsive and flexible unlike Twitter's constrained Tweet Box:

Style without Placeholders

In order to get our final target render to match Twitter, we simply need to *constrain* it by updating our `content-tweet-box` class. Updating it from:

```
.content-tweet-box {
    padding: 0.5rem 1rem;
}
```

to:

```
.content-tweet-box {
    background-color: #FFFFFF;
    padding: 0.5rem 1rem;
    height: 118px;
    width: 598px;
}
```

while additionally changing the `background-color` of the entire `body` with:

```
body {
    background-color: #EEEEEE;
}
```

finally gives us our target render:

Target Render

For comparison, here is the original screenshot of Twitter's Tweet Box UI that we referenced at the beginning of this chapter:

Tweet Box UI Reference

Our HTML and CSS solutions are different from the original Tweet UI code as we used our 80/20 subset, *but we get the same visual effect*. The above exercise:

1. Reinforces the power of 80/20
2. Exemplifies iteration and the *Work. Right. Better.* approach
3. Provides a detailed step-by-step process for converting a static design into a dynamic and interactive one

We can go much further regarding its dynamic and interactive nature however. Only the browser defaults of responsive layout and button hover interactivity exists currently. We cannot type in the Tweet Box and and nothing else it interactive. This leads us to our final reconstruction step.

#

Reconstruct Behavior

This process consists of three core substeps:

1. Change - *Determine what can change at execution time*
2. Depend - *Determine the change dependencies*
3. Component - *Create components using these facts*
    - Shell - *Code the component shell*
    - Interface - *Code the component interface*
    - Implementation - *Code the component implementation*

#

Change

For this substep the goal is to define what can *change* at execution time. This is determined from general experience, specific product use, or through mental simulation. The latter case is especially true when designing something new from scratch. In either case, the goal is to list what will change at execution time based on the aforementioned *input triggers*. As a reminder these triggers are:

1. User interaction (tap, click, hover, gesture, voice, etc.)
2. Environment (layout resizing, operating system, device sensors, etc.)
3. Time (delays, schedules, etc.)

For the Tweet UI, our change list includes the following:

1. What's happening? focused state & visibility
2. Tweet text
3. Progress bar Tweet character limit
4. Enabled vs. Disabled state of Tweet button
5. Tweet media button popups
6. Add Tweet button popup

Moving forward we'll ignore the latter two popup related changes to scope down this chapter's length. We will focus solely on the Tweet message text and its impact on the rest of the UI. This subset focus will suffice as it encompasses all four combinations of state changes that can occur in interactive applications. These are:

1. Binary & Source
2. Binary & Derived
3. Many & Source
4. Many & Derived

You can likely infer the combined meaning of each, but we'll detail them below soon.

Now that we've defined what *can change* at execution time we may begin to define the *dependencies* of those changes.

#

Depend

Inherent in change is the implied fact that there are *states*. Programming and Visual Design - Texture and State first introduced us to states in the context of coding:

> Texture when applied to a shape or a form gives it a richer quality. This richness is exemplified as a sense of time, where a texture is aged and weathered for example. Similarly, a function or object with state gives it a richer quality, a sense of time as well.

If something can change, it is at the very least in *this* state or *that* state at any moment in time. What does this remind you of?

Yup, binary. Defining and intuiting what is binary or not takes time to get used to, but it is a straightforward concept to apply in UIs. In fact, there are two questions to ask after determining the change list:

- What has *only two states*? (Binary)
- What has *three or more states*? (Many)

We can ask ourselves these questions for each item in our change list to get an answer. Specifically for our Tweet UI change list we get:

1. What's happening? focused state (Binary) & visibility (Binary)
    - Focused
    - Unfocused
    - Visible
    - Invisible
2. Tweet text (Many)
    - Zero characters
    - One character
    - Two characters
    - Etc.
3. Progress bar Tweet character limit (Many)
    - Zero percent
    - One percent
    - Two percent
    - Etc.
4. Enabled vs. Disabled state of Tweet button (Binary)

    - Enabled
    - Disabled

Using the below refresher from 80/20 JavaScript - Types & Forms section, try to determine the best-match *value type* to represent the state of each item in the change list.

1. Primitive Values
    - null (`null`)
    - undefined (`undefined`)
    - Boolean (`true` & `false`)
    - Number (`360`)
    - String (`"one or more characters wrapped in double quotes"` or `'single quotes'`)
2. Complex Values
    - Object (`{}` & `[]`)

Using this information, what is the best-match value type for each item in our change list? The result we're looking for is:

1. What's happening? focused state (`Boolean`/Binary) & visibility (`Boolean`/Binary)
2. Tweet text (`String`/Many)
3. Progress bar Tweet character limit (`Number`/Many)
4. Enabled vs. Disabled state of Tweet button (`Boolean`/Binary)

After determining the value type of each item in the change list, we can proceed more informed. In fact we can further help define the change dependencies through identifying the change list item's *state type*. There are only two:

1. Source (explicit state)
2. Derived (implied state)

We have not explored the two *state types* as they were not important until now. They will help us implement our behavior code with less duplication. Remember Don't Repeat Yourself? This is where *derived* state shines.

Each primitive or complex value in an application is either source or derived. That fact depends solely on the expected behavior of the application in question. In the context of the Tweet Box UI that means we need to determine which (if any) of our change list item's have their state set explicitly (source) or should be implied (derived). Thinking in terms of derived state takes time to get used to, but the better you get at it the less redundant your code can become. Thankfully there is one simple question to ask to help determine if state is source or derived:

> Can the state of something be defined in terms of the state of something else?

Before reading on, ask the above question against each of the four change list items and their defined value types. Which values can be derived? This will be challenging as it takes general experience, specific product use, and mental simulation, but try your best. Here is what we're looking for:

1. What's happening? focused state (`Boolean`/Binary & Source) & visibility (`Boolean`/Binary & Derived)
2. Tweet text (`String`/Many & Source)
3. Progress bar Tweet character limit (`Number`/Many & Derived)
4. Enabled vs. Disabled state of Tweet button (`Boolean`/Binary & Derived)

Ask yourself, what makes the latter two derived? Using our helper question for each, what is the *something else* that would make it so? Feel free to think through this before moving on as we reveal the answers below.

To complete this substep and empower us in the Component substep we must define our change dependencies by noting their input triggers and source vs. derived state type. Here we go:

1. What's happening? focused state (`Boolean`/Binary & Source) & visibility (`Boolean`/Binary & Derived)
    - Focused
        - User interaction (click/tap in)
        - User interaction (tab keyboard in)
    - Unfocused
        - User interaction (click/tap out)
        - User interaction (tab keyboard out)
    - Visible
        - Derived (from Tweet text `String` length)
    - Invisible
        - Derived (from Tweet text `String` length)
2. Tweet text (`String`/Many & Source)

    - Zero characters
        - User interaction (key down/up)
        - User interaction (cut/paste)
    - One character
        - User interaction (key down/up)
        - User interaction (cut/paste)
    - Two characters
        - User interaction (key down/up)
        - User interaction (cut/paste)
    - Etc.
        - User interaction (key down/up)
        - User interaction (cut/paste)
3. Progress bar Tweet character limit (`Number`/Many & Derived)
    - Zero percent
        - Derived (from Tweet text `String` length)
    - One percent
        - Derived (from Tweet text `String` length)
    - Two percent
        - Derived (from Tweet text `String` length)
    - Etc.
        - Derived (from Tweet text `String` length)
4. Enabled vs. Disabled state of Tweet button (`Boolean`/Binary & Derived)
    - Enabled
        - Derived (from Tweet text `String` length)
    - Disabled
        - Derived (from Tweet text `String` length)

Now we have what we need to inform how we structure our custom *components*. Let's to this.

#

Component

We first described *components* in the Programming & Visual Design - Constructs and Components section:

> In programming, components embody the elements and design patterns in an effort to encapsulate specific functionality and ways of communicating with other components.

Additionally in 80/20 JavaScript - New Operator, we learned that objects are very useful containers for such a thing:

> ... use the `new` operator and think of it as your helper for getting unique instances of built-in, third-party, and custom types of Objects

Since we are creating non-built-in and non-third-party components, they must each be custom. Since there are four change list items, you might think we need four components. Sometimes the relationship is one-to-one like that, but sometimes it is not. In this particular case we can merge the first two into one component.

1. `TweetText` (What's happening? and Tweet text)
2. `ProgressBar`
3. `TweetButton`

By following the same useful pattern that was first demonstrated in the `Artboard` example. We can code the *shell*.

#

Shell

Defining a shell for each component is extremely simple and consists of the following:

1. Define a `function` and name it
2. Define its `api`
3. `return` its `api`

Our three components become:

```
function TweetText() {
    var api = {};
    return api;
}

function ProgressBar() {
    var api = {};
    return api;
}

function TweetButton() {
    var api = {};
    return api;
}
```

Admittedly this isn't very useful as the `api` of each component is empty. Additionally the `function` doesn't do anything other than `return` this empty `api`. We'll resolve this shortly.

Since we already completed the *Change* and *Depend* substeps we can use their results to inform our component design. We will use Zoom Level 1 to do so as we don't need to worry about the implementation of the code yet. Only then would zoom levels two and three become useful.

#

Interface

Once we have our component shells, we need to think through the API of each component. Put another way, we need to define how the components:

- Manage their state
  - Get updated values
  - Set updated values
- Communicate state changes
  - Dispatch events and/or values

It's worth noting that frameworks such as Vue.js, React, and Angular exist to help with massively simplifying and easing the component creation, destruction, and state management processes of applications both small and large. I highly recommend using one of them in the future, but the scope of this book is to communicate how to use 80/20 JavaScript in its vanilla form. Having a deeper understanding of coding generally and JavaScript specifically will only help you when using frameworks in the future.

Onward. The first step we can take is by identifying certain HTML nodes so that we can talk to them with JavaScript. As you may recall from earlier the `id` attribute helps us here. So our HTML that was:

```
<p class='empty-space-grow content-tweet-text'>What's happening?</p>
```

becomes:

```
<p id='tweet-text' class='empty-space-grow content-tweet-text'>
    What's happening?
</p>
```

We do this for each node that we want to update at execution time, trigger events on, and/or listen to events from. This results in the `id='progress-bar'` and `id='tweet-button'` additions as well.

With this effort we can update our components to each have an approach for getting the reference to their element of interest. Remember, each element's corresponding JavaScript object has *its own API*. We will use this fact in the future to get more done with less.

```
function TweetText() {
    var element = document.getElementById('tweet-text');
    var api = {};

    return api;
}

function ProgressBar() {
    var element = document.getElementById('progress-bar');
    var api = {};
```

```
    return api;
}

function TweetButton() {
    var element = document.getElementById('tweet-button');
    var api = {};

    return api;
}
```

Using the work from the previous substeps we can iterate on our components by starting to define the `function` and `var`iable names to associate with the input triggers and state respectively.

This is just a first pass. Coding is an iterative process as you may recall from Refactor Early and Often:

> It can even be difficult to name identifiers well (believe it or not this is one of the more difficult aspects of programming). Thankfully the code is not set in stone. It is easy to move, change, and rename. This is refactoring.

With this in mind we can update our existing change list by appending proposed `function` and `var`iable names.

`TweetText`:

- Focused vs. Unfocused
  - Focused `isFocused` (assigned `true`)
    - User interaction (click/tap in) `onFocusIn()`
    - User interaction (tab keyboard in) `onFocusIn()`
  - Unfocused `isFocused` (assigned `false`)
    - User interaction (click/tap out) `onFocusOut()`
    - User interaction (tab keyboard out) `onFocusOut()`
  - Visible `isVisible` (assigned `true`)
    - Derived (from Tweet text `String` length)
  - Invisible `isVisible` (assigned `false`)
    - Derived (from Tweet text `String` length)
- Visible vs. Invisible
  - Visible `isVisible` (derived `true`)
    - Derived (from Tweet text `String` length) `text` of `TweetText`
  - Invisible `isVisible` (derived `false`)
    - Derived (from Tweet text `String` length) `text` of `TweetText`
- Tweet text
  - Zero characters `text` (no input)
    - User interaction (key down/up) `onInputChange()`
    - User interaction (cut/paste) `onInputChange()`
  - One character `text` (one input)
    - User interaction (key down/up) `onInputChange()`
    - User interaction (cut/paste) `onInputChange()`
  - Two characters `text` (two inputs)
    - User interaction (key down/up) `onInputChange()`
    - User interaction (cut/paste) `onInputChange()`
  - Etc. `text` (you get it)
    - User interaction (key down/up) `onInputChange()`
    - User interaction (cut/paste) `onInputChange()`

`ProgressBar`:

- Completion percentage
  - Zero percent `percent` (0/280 characters)
    - Derived (from Tweet text `String` length) `onInputChange()` of `TweetText`
  - One percent `percent` (3/280 characters)
    - Derived (from Tweet text `String` length) `onInputChange()` of `TweetText`
  - Two percent `percent` (6/280 characters)
    - Derived (from Tweet text `String` length) `onInputChange()` of `TweetText`
  - Etc. `percent` (you get it)
    - Derived (from Tweet text `String` length) `onInputChange()` of `TweetText`

`TweetButton`:

- Enabled vs. Disabled
  - Enabled `isEnabled` (derived `true`)
    - Derived (from Tweet text `String` length) `text` of `TweetText`
  - Disabled `isEnabled` (derived `false`)
    - Derived (from Tweet text `String` length) `text` of `TweetText`

After this first pass, we can update our components using the `function` and `var`iable names we identified.

```
function TweetText() {
    var isFocused;
    var isVisible; // Derived via `text` of `TweetText`. We will learn to do this later.
    var text;
    var element = document.getElementById('tweet-text');
    var api = {}

    function onFocusIn() {}
    function onFocusOut() {}
    function onInputChange() {}

    return api;
}

function ProgressBar() {
    var percent; // Derived via `onInputChange()` of `TweetText`. We will learn to do this later.
    var element = document.getElementById('progress-bar');
    var api = {};

    return api;
}

function TweetButton() {
    var isEnabled; // Derived via `text` of `TweetText`. We will learn to do this later.
    var element = document.getElementById('tweet-button');
    var api = {};

    return api;
}
```

And with that we have the interfaces for our components defined. They will likely change as we iterate but they are in a great spot for us to start our *implementation*.

#

Implementation

APIs let us do more with less. These APIs can be provided by the language, environment, or a third-party (including ourselves). In any case, they allow us to do more with less.

As you continue to learn what is possible in HTML, CSS, and JavaScript, you can prevent reinventing the wheel through your knowledge of APIs. You don't need to memorize them or anything, but it helps *to know they exist*. Now is a good time to reflect on a recommendation from earlier in the Non-Reserved Keywords - Environment section:

> I do however recommend exploring the list of all the web APIs sometime. The effort enables you to grasp the big picture of what is possible by default in the browser.

This recommendation was made in the context of JavaScript, but the same holds true for HTML and CSS. There are numerous HTML element types and CSS properties that are truly empowering. In the context of our Tweet Box UI for example we can and will step outside our 80/20 element coverage as the `<textarea>` element is great for allowing a user to edit text like a Tweet.

It's worth mentioning that the `<textarea>` will fall a bit short of the actual Tweet Box text box. This is the case because it does not support nested elements like `<span>`s and `<a>` that Twitter uses to properly implement at tags (`@`) and hashtags (`#`). If there is enough interest in this complex implementation I will add a bonus chapter of advanced HTML, CSS, and JavaScript to this book.

For now, the `<textarea>` will get us 80/20 of the features with minimal effort. Our update will consist of simply swapping out the element type and adding a `placeholder` attribute.

```
<p id='tweet-text' class='empty-space-grow content-tweet-text'>
    What's happening?
```

```
</p>
```

becomes:

```
<textarea
    id='tweet-text'
    class='empty-space-grow content-tweet-text'
    placeholder="What's happening?"></textarea>
```

An additional CSS update makes it so the `<textarea>` can grow its `height` automatically, prevent manual `resize`ing, and visually remove an undesirable `border` and `outline`.

```
.content-tweet-box {
  height: 100%;
  /* Other styles not shown for brevity */
}

.content-tweet-text {
  border: none;
  outline: none;
  resize: none;
  /* Other styles not shown for brevity */
}
```

With just these small changes, we can now add, edit, cut, copy, paste, or delete text. Nice!

Tweet Box with Textarea

With this addition of the `<textarea>` we can now make another pass on our interface and implementation by leveraging its text changes to derive updates for our components. The first iterative update is to encapsulate our three components in a new component. We do this to better manage, nest, and communicate code relationships. Since the `TweetText`, `ProgressBar`, and `TweetButton` can be grouped and parented by a new `TweetBox` component, that's what we'll do. We will do so using our previously used interface substep approach:

```
function TweetBox() {
    var tweetText;
    var progressBar;
    var tweetButton;
    var api = {};

    function TweetText() {
        var isFocused;
        var isVisible; // Derived via `text` of `TweetText`. We will learn to do this later.
        var text;
        var element = document.getElementById('tweet-text');
        var api = {}

        function onFocusIn() {}
        function onFocusOut() {}
        function onInputChange() {}

        return api;
    }

    function ProgressBar() {
        var percent; // Derived via `onInputChange()` of `TweetText`. We will learn to do this later.
        var element = document.getElementById('progress-bar');
        var api = {};

        return api;
    }

    function TweetButton() {
        var isEnabled; // Derived via `text` of `TweetText`. We will learn to do this later.
```

```
        var element = document.getElementById('tweet-button');
        var api = {};

        return api;
    }

    return api;
}
```

If you have been paying close attention, you may have realized that although we're defining our interface and implementation, the code in our functions won't actually run yet. This is because none of our `function`s have *been called*. Let's fix this now:

```
function TweetBox() {
    var tweetText = new TweetText();
    var progressBar = new ProgressBar();
    var tweetButton = new TweetButton();
    var api = {};

    function TweetText() {
        var isFocused;
        var isVisible; // Derived via `text` of `TweetText`. We will learn to do this later.
        var text;
        var element = document.getElementById('tweet-text');
        var api = {}

        function onFocusIn() {}
        function onFocusOut() {}
        function onInputChange() {}

        return api;
    }

    function ProgressBar() {
        var percent; // Derived via `onInputChange()` of `TweetText`. We will learn to do this later.
        var element = document.getElementById('progress-bar');
        var api = {};

        return api;
    }

    function TweetButton() {
        var isEnabled; // Derived via `text` of `TweetText`. We will learn to do this later.
        var element = document.getElementById('tweet-button');
        var api = {};

        return api;
    }

    return api;
}

var tweetBox = new TweetBox();
```

Now all four functions run and thus their code gets executed. Still not much happens outside various `var`iable assignments. Since the `api`s of each are all empty, not much is going on in the application. Let's change that.

As we've previously defined, we have input triggers that we want to listen to. In order to do this, we'll update `TweetText`'s `api` so that its useful to the other components. Now is the time to starting thinking in zoom level 2 and 3.

```
function TweetBox() {
    var tweetText = new TweetText();
    var progressBar = new ProgressBar();
    var tweetButton = new TweetButton();
    var api = {};

    tweetText.element.addEventListener('input', onTweetTextInput);

    function onTweetTextInput(event) {
        console.log('The textarea change is: ' + event.data);
    }

    function TweetText() {
        var isFocused;
        var isVisible; // Derived via `text` of `TweetText`. We will learn to do this later.
```

```
        var text;
        var element = document.getElementById('tweet-text');
        var api = { element: element }

        function onFocusIn() {}
        function onFocusOut() {}
        function onInputChange() {}

        return api;
    }

    function ProgressBar() {
        var percent; // Derived via `onInputChange()` of `TweetText`. We will learn to do this later.
        var element = document.getElementById('progress-bar');
        var api = {};

        return api;
    }

    function TweetButton() {
        var isEnabled; // Derived via `text` of `TweetText`. We will learn to do this later.
        var element = document.getElementById('tweet-button');
        var api = {};

        return api;
    }

    return api;
}

var tweetBox = new TweetBox();
```

Take note of the `onTweetTextInput` and that it gets called on the input trigger. Right now we're only outputting the change to the `console`, but as we know from our previous substeps, we want to use this information to `derive` values for our `ProgressBar` and `TweetButton`. To do this we'll focus on `onTweetTextInput`s implementation and additionally update the `api`s of both the `ProgressBar` and `TweetButton`.

```
function onTweetTextInput(event) {
    var allTheText = event.target.value;
    progressBar.update(allTheText);
    tweetButton.update(allTheText);
}
```

As a result of this implementation, we additionally need `update` implementations. The iterative updates result in:

```
function TweetBox() {
    var tweetText = new TweetText();
    var progressBar = new ProgressBar();
    var tweetButton = new TweetButton();
    var api = {};

    tweetText.element.addEventListener('input', onTweetTextInput);

    function onTweetTextInput(event) {
        var allTheText = event.target.value;
        progressBar.update(allTheText);
        tweetButton.update(allTheText);
    }

    function TweetText() {
        var isFocused;
        var text;
        var element = document.getElementById('tweet-text');
        var api = { element: element }

        function onFocusIn() {}
        function onFocusOut() {}
        function onInputChange() {}

        return api;
    }

    function ProgressBar() {
        var percent; // Derived via `onInputChange()` of `TweetText`. We will learn to do this later.
        var element = document.getElementById('progress-bar');
        var api = { update: update };
```

```
        function update(text) {
            console.log('Derive percent using text argument value');
        }

        return api;
    }

    function TweetButton() {
        var isEnabled; // Derived via `text` of `TweetText`. We will learn to do this later.
        var element = document.getElementById('tweet-button');
        var api = { update: update };

        function update(text) {
            console.log('Derive isEnabled using text argument value');
        }

        return api;
    }

    return api;
}

var tweetBox = new TweetBox();
```

Since updating to our `<textarea>` element, we don't need to derive the `isVisible` state of the `TweetText` as that functionality is built-in to it. When implementing the actual Twitter Tweet Box component, we'd still have to manage this state however. For now, we do not and that's why it was removed in the snippet above.

Now let's replace our temporary `console.log` statement with our desired `update` implementation in the `ProgressBar`.

```
function update(text) {
    var valueWithDecimal = (text.length / 280) * 100;
    percent = Math.ceil(valueWithDecimal);

    render();
}
```

It's the `TweetButton`'s turn.

```
function update(text) {
    var isAboveMinimum = text.length > 0;
    var isBelowMaximum = text.length < 281
    isEnabled = isAboveMinimum && isBelowMaximum;

    render();
}
```

Since we want these updates to be communicated visually, we also need to tell the browser. We do this with a `render function` that leverages its component's `element` reference in some way. If a component doesn't need to be reflected visually, then it wouldn't need a reference to an element. Just something to think about when you are defining components in the future.

For the `ProgressBar` we take a shortcut in the `render function` and display text instead of updating the SVG ring. I'll leave this as a challenge to explore. Here is the component in its entirety:

```
function ProgressBar() {
    var percent;
    var element = document.getElementById('progress-bar');
    var api = { update: update };

    function update(text) {
        var valueWithDecimal = (text.length / 280) * 100;
        percent = Math.ceil(valueWithDecimal);

        render();
    }

    function render() {
        element.innerText = percent;
    }

    return api;
}
```

For the `TweetButton` we add the familiar `halve-opacity` CSS class and either `add` or `remove` it at execution time. Here is the component in its entirety:

```
function TweetButton() {
    var isEnabled;
    var element = document.getElementById('tweet-button');
    var api = { update: update };

    function update(text) {
        var isAboveMinimum = text.length > 0;
        var isBelowMaximum = text.length < 281
        isEnabled = isAboveMinimum && isBelowMaximum;

        render();
    }

    function render() {
        element.disabled = !isEnabled;

        if(isEnabled) {
            element.classList.remove('halve-opacity');
        } else {
            element.classList.add('halve-opacity');
        }
    }

    return api;
}
```

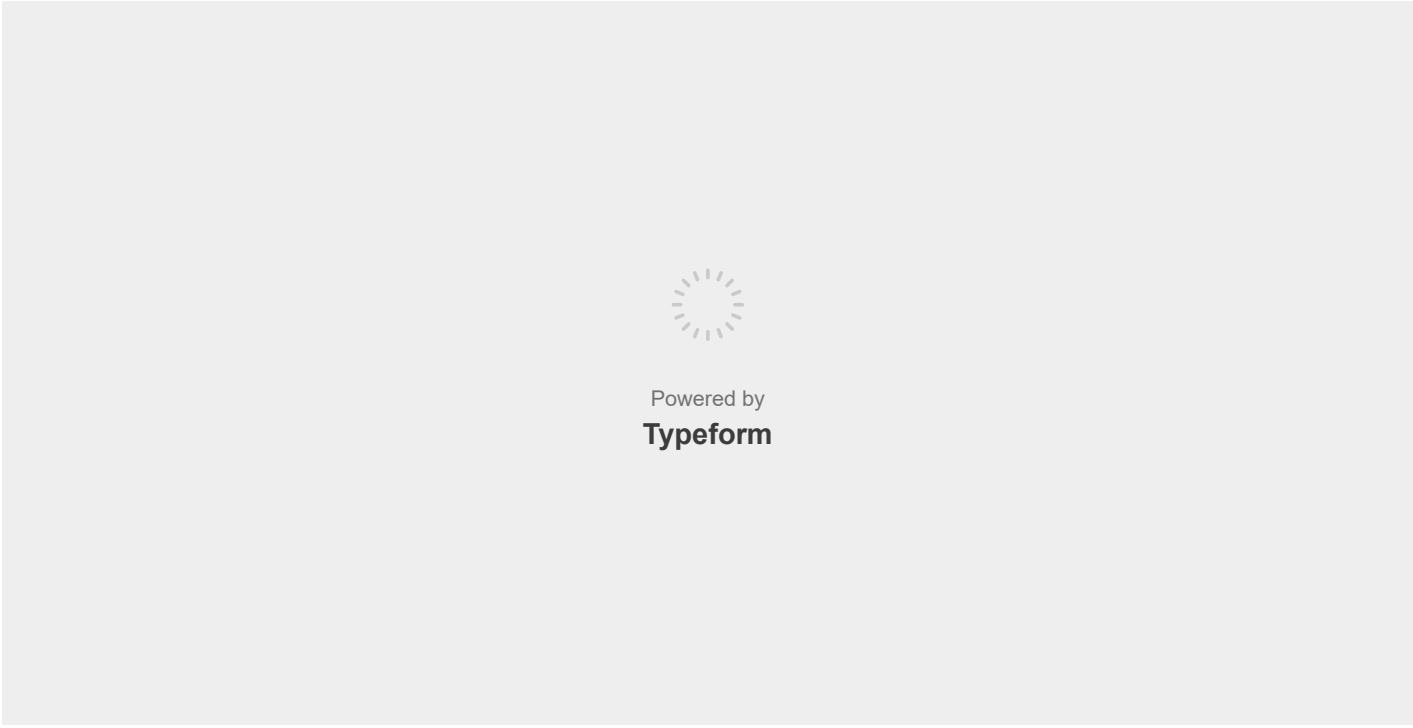
With these two component updates we've iterated our design closer toward Twitter's actual experience. Here is the updated interactive sample:

Tweet Box Using Derived Values

Though we could continue iterating to gradually make our 80/20 `TweetBox` component more robust and feature complete, we will stop here.

Throughout this chapter we learned a lot about deconstructing and reconstructing designs. In addition to defining and exploring a generalized step-by-step process, we used it in practice against the Twitter Tweet Box UI. It is important to stress that this same process is useful in not only existing designs, but the ones you dream up and manifest as you grow as a designer and coder.

Thank you for reading and good luck on your continued journey!



Chapter 5
80/20
JavaScript