1

A BOOK APART

GET READY FOR CSS GRID LAYOUT

RACHEL ANDREW

FOREWORD BY
ERIC MEYER

BRIEFS

MORE FROM A BOOK APART BRIEFS

Pricing Design
Dan Mall

Visit abookapart.com for our full list of titles.

Publisher: Jeffrey Zeldman
Designer: Jason Santa Maria
Executive Director: Katel LeDû
Editor: Caren Litherland
Technical Editor: Paul Lloyd
Copyeditor: Lisa Maria Martin
Compositor: Rob Weychert
Ebook Producer: Ron Bilodeau

ISBN: 978-1-9375572-7-0

A Book Apart
New York, New York
http://abookapart.com

10 9 8 7 6 5 4 3 2 1

TABLE OF CONTENTS

FOREWORD

From the very beginning of CSS, there has been a giant, layout-shaped hole at its center.

We filled it with table markup, which was never meant for layout. After that came floats, which were also never meant for layout; we only used them because `clear` existed, so we could push footers to the bottom of the page. (Yes, pages used to have bottoms. Kids, ask your parents about finite scrolling.) For a brief period, we played with positioning, which actually *was* meant for layout, but was too incompletely designed to effectively serve that role.

And that's where things have stood for well more than a decade: hacking our way to layouts with floats. A useful and often ingeniously adapted hack, but a hack nonetheless. It's little wonder that CSS has a bad reputation in certain quarters. If your presentation system has to be hacked just to lay out the basic page elements...

That time is now drawing to a close. Flexbox came first, and is already upending what we think we know about layout. Close behind Flexbox came Grid Layout. If you love Flexbox, there may be no words to describe the emotions you'll feel for Grid Layout. Grid Layout is to Flexbox as PNG is to BMP, and then some. It contains power we have scarcely dreamed of, let alone dared to hope we might use.

I might be just a little excited about Grid Layout. But then, after such a long wait, how could I not be?

As excited as I am about Grid Layout itself, though, I'm even more excited that Rachel Andrew has written this book and that you have it in front of you right now. This potent little tome packs a great look at everything grids currently have to offer. And that's no surprise coming from Rachel, who has long been one of the clearest and sharpest CSS educators around. This is a text I'll refer back to as I practice using grids. I suspect you will, too.

Grids have been a long, long time coming, but they're finally here. Enjoy your newfound power, and join me in thanking Rachel for showing us how to wield it.

—Eric Meyer

INTRODUCTION

When I began working on the web in 1996, the only real skill a front-end developer had to master was chopping up images into tiny bits and reassembling them into a table to create a layout.

Netscape 4 still held a huge market share when I started using CSS for layout. The browser's implementation of absolute positioning was so poor that when a user resized their screen, all of the positioned elements would stack up in the top left corner. I've watched CSS evolve from a simple single specification—concerned primarily with changing text colors and adding borders to things—to the increasingly complex language it is today.

Along the way, I've witnessed browser wars and, during my time as a Web Standards Project member, have encouraged browser and tool vendors alike to innovate through the standards process. We can now see that process playing out in many of the specifications currently wending their way through the W3C.

One such specification, CSS Grid Layout, is the subject of this little book. The specification debuted under this name as a proposal by Microsoft in April 2011 (http://bkaprt.com/cgl/00-01/). This early version of the specification appeared in Internet Explorers 10 and 11 and has since been adopted by the W3C. The current editors represent Google, Mozilla, and Microsoft; and, as Level 1 of the specification nears completion, new Editor's Drafts are appearing apace.

In addition to the early (and now outdated) Microsoft implementation, there is an excellent and robust implementation of most of the Level 1 specification in the Blink and Webkit rendering engines. It's a very different world from the one in which I learned my craft!

I began experimenting with CSS Grid Layout as soon as I discovered the IE10 implementation. For several years I've been frustrated that layout hasn't advanced much, despite our ability to round corners, create drop shadows, use fonts, and even animate things in CSS. We now have better ways to cope with floating and positioning elements, and our browsers are less buggy—yet the techniques we use for layout are not far

removed from the ones we used in the early days of CSS. As soon as I started experimenting with Grid Layout, I could see its potential. I really believe that Grid Layout is the layout method we've been waiting for.

WHAT IS CSS GRID LAYOUT?

THE CSS GRID LAYOUT MODULE is a new CSS module that defines a two-dimensional grid layout system. Once a grid has been established on a containing element, the children of that element can be placed into slots on a flexible or fixed layout grid. The grid can be redefined using media queries, rendering the source order of the child elements unimportant. This makes CSS Grid Layout an incredibly powerful tool—one that I'm excited about seeing in our browsers.

Rather than talk about CSS Grid Layout (or just plain "Grid," as I will call it often throughout this text) in abstract terms, I'll demonstrate its functionality as implemented in the Blink rendering engine for Chrome. First, activate Grid by entering the following line in your browser address bar:

```
chrome://flags/#enable-experimental-web-platform-
  features
```

Then, turn on the Enable Experimental Web Platform Features flag and restart Chrome.

Grid will likely be released in Blink later in 2015. Although, as I mentioned, there is an early implementation of the module in Internet Explorers 10 and 11, the specification has moved on. The IE implementation now differs significantly from what we will be discussing.

WHY EXPLORE THIS EARLY-STAGE SPECIFICATION?

It could be argued that CSS Grid Layout is too far off in terms of good browser support for us to really take much interest in it. But I believe it is vitally important that we, as developers and designers, get involved early. Unless developers engage with specifications as they move through the W3C process and become implemented in browsers, how can we offer real feedback to those who are doing this work? What right will we

have to state that the result isn't fit for our purposes if we ignore it until the specification becomes a W3C Recommendation and is implemented in a number of browsers?

GRID BASICS

In this first chapter, I want to use some simple examples to provide a rundown of the essential concepts of the CSS Grid Layout Module. Grid is a very flexible module, so there are a number of ways to use it. In the following chapters, we'll look at some more "real-world" examples, building on what I describe here.

Defining a grid

A grid is defined using a new value of the display property, `display: grid`.

In my HTML markup, I want to create a grid on the wrapper and position the child elements on that grid.

```
<div class="wrapper">
  <div class="box a">A</div>
  <div class="box b">B</div>
  <div class="box c">C</div>
  <div class="box d">D</div>
  <div class="box e">E</div>
  <div class="box f">F</div>
</div>
```

In my CSS, I start by declaring a grid on the element with a class of `.wrapper`:

```
.wrapper {
  display: grid;
}
```

Next, I need to describe what the grid looks like. Grids have rows and columns, which the CSS Grid Layout Module gives us new properties to describe:

```
grid-template-rows
grid-template-columns

   .wrapper {
   display: grid;
   grid-template-columns: 100px 10px 100px 10px »
     100px;
   grid-template-rows: auto 10px auto;
 }
```

Here I have created a grid with three 100-pixel-wide columns separated by 10-pixel gutter columns. There are three rows specified, two set to `auto`, so they will expand to hold whatever amount of content is put inside them. They are separated by a 10-pixel gutter row.

If you take a look at your page after you've declared a grid, you'll find that the child elements have made an attempt to place themselves on the grid (**FIG 1.1**). They do this according to the grid's auto-placement rules, which simply fill each cell in turn, so that children have been placed in the gutter columns. Obviously, this is not ideal!

We can sort this out by deliberately positioning our items on the new grid. The simplest way to do that is to use line-based placement. I can use the following rules to place an element whose class is `.a` into the first cell of the grid.

```
.a {
  grid-column-start: 1;
  grid-column-end: 2;
  grid-row-start: 1;
  grid-row-end: 2;
}
```

Code example: http://bkaprt.com/cgl/01-01/

This, and all of the examples in this book, can be found on GitHub (http://bkaprt.com/cgl/01-02/). I'll reference the file below each example.

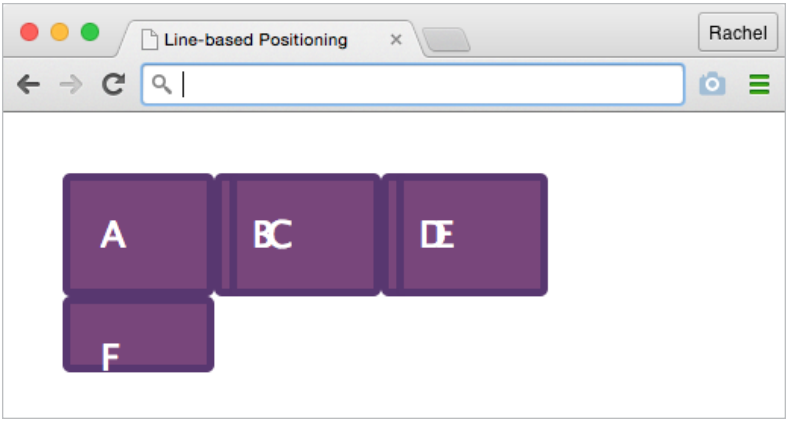

FIG 1.1: The browser will display child elements of a grid according to the grid's auto-placement rules. In general, however, you will want to position the items yourself.

We can also express this in shorthand by using the `grid-column` and `grid-row` properties, in which the first value represents the column or row *start* and the second value represents the column or row *end*.

```
.a {
  grid-column: 1 / 2;
  grid-row: 1 / 2;
}
```

Code example: http://bkaprt.com/cgl/01-03/

We can use an even shorter shorthand with the `grid-area` property. Here the order of values is:

```
grid-row-start
grid-column-start
grid-row-end
grid-column-end
```

This gives us:

```
.a {
  grid-area: 1 / 1 / 2 / 2;
}
```

Code example: http://bkaprt.com/cgl/01-04/

Personally, I find the shorter shorthand a little difficult to read. For clarity, I prefer the `grid-column` and `grid-row` shorthand.

By drawing a box around the area we want our content to go into, line-based placement creates a grid area. The grid we defined on our wrapper creates six 100-pixel-wide grid cells, and I can place the child elements into them using the following rules. This will appear in the browser as a set of boxes with gutters between them (**FIG 1.2**).

```
.a {
  grid-column: 1 / 2;
  grid-row: 1 / 2;
}

.b {
  grid-column: 3 / 4;
  grid-row: 1 / 2;
}

.c {
  grid-column: 5 / 6;
  grid-row: 1 / 2;
}

.d {
  grid-column: 1 / 2;
  grid-row: 3 / 4;
}
```

```
.e {
  grid-column: 3 / 4;
  grid-row: 3 / 4;
}

.f {
  grid-column: 5 / 6;
  grid-row: 3 / 4;
}
```

Code example: http://bkaprt.com/cgl/01-03/

A grid area can span as many individual grid cells as required. You just need to specify the line where the content will start and the line where it will end.

Now, I'll use the same declared grid, but will only place four child elements onto it (**FIG 1.3**).

```
.a {
  grid-column: 1 / 4;
  grid-row: 1 / 2;
}

.b {
  grid-column: 5 / 6;
  grid-row: 1 / 4;
}

.c {
  grid-column: 1 / 2;
  grid-row: 3 / 4;
}

.d {
  grid-column: 3 / 4;
  grid-row: 3 / 4;
}
```

Code example: http://bkaprt.com/cgl/01-05/

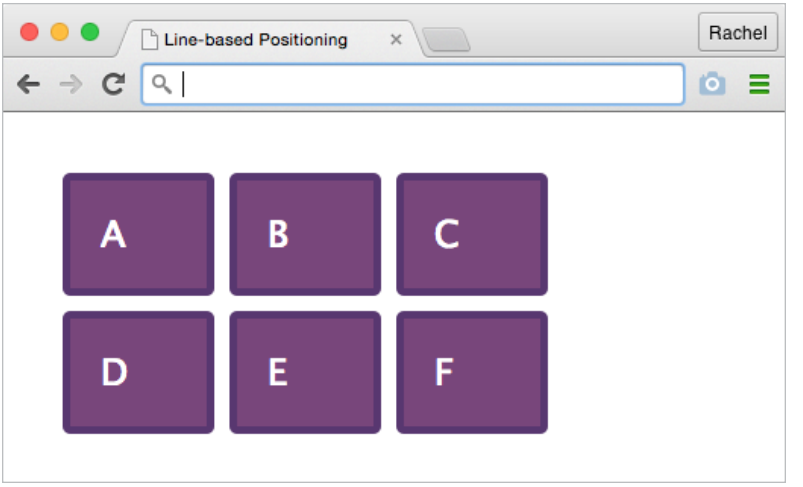

FIG 1.2: A simple layout of equally sized boxes placed onto a grid.

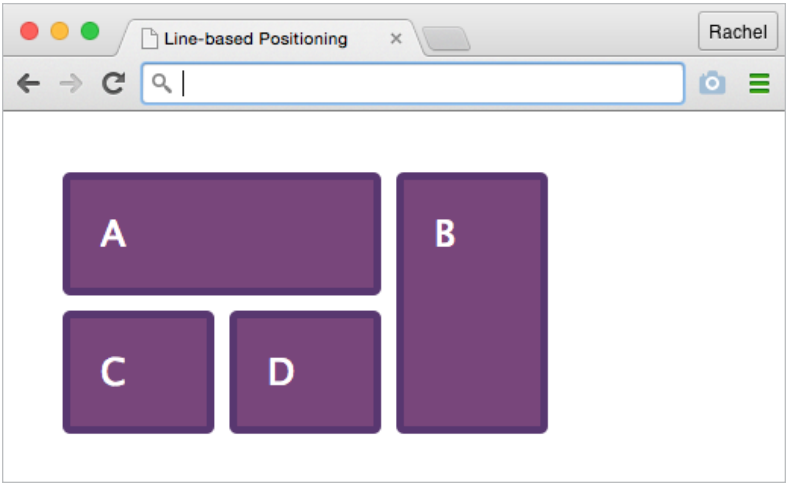

FIG 1.3: Spanning with line-based placement.

Note that the source order of these child elements doesn't matter. With Grid, you can order your source in whatever way makes sense for the document, and then display it in whatever way makes sense visually for any viewport size.

GRID TERMINOLOGY

Before going any further, let's take a few moments to understand some of the terminology used when talking about Grid Layout.

Grid lines

Grid lines make up the grid and can be horizontal or vertical. They can be referred to by number, as I have done so far, but they can also be named.

Grid track

A *grid track* is the space between two grid lines. It can be either horizontal or vertical.

In the examples shown in **FIGS 1.1–1.3**, we have grid tracks representing content cells and grid tracks representing gutters. As far as the grid is concerned, these are the same thing. So when positioning using numbered lines, remember that the gutter lines exist and be sure to include them when working out where content starts and ends.

Grid cell

A *grid cell*—the space between four grid lines—is the smallest possible unit on the grid. Conceptually, it is just like a table cell.

Grid area

A *grid area* is any area of the grid bound by four grid lines. It can contain a number of grid cells.

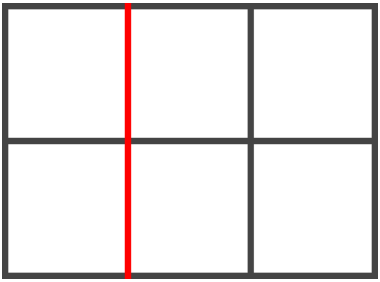

FIG 1.4: The highlighted grid line is column line 2.

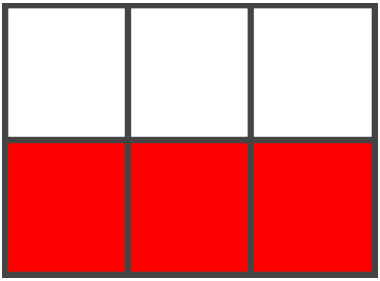

FIG 1.5: I have highlighted the track between row lines 2 and 3.

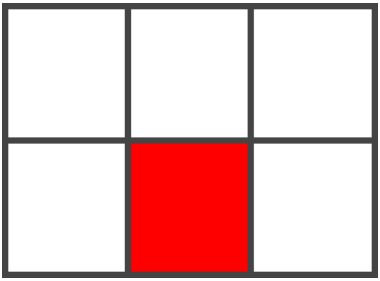

FIG 1.6: The highlighted grid cell in this image is between row lines 2 and 3 and column lines 2 and 3.

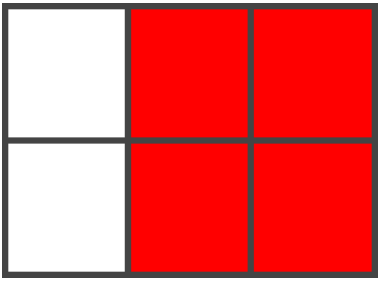

FIG 1.7: The highlighted grid area in this image is between row lines 1 and 3 and column lines 2 and 4.

Grid gutters

The examples in this little book all achieve gutters by way of a grid track that serves as a gutter between tracks used for content. This is a valid approach, and is, in fact, necessary if you want to create a complex grid using different gutter widths.

As I was preparing this book, a `grid-column-gap` property was added to the Level 1 specification. This property behaves much like `column-gap` in multiple-column layouts. Also, for Grid we have `grid-row-gap` to create spacing between rows. These properties have a shorthand of `grid-gap`, which allows you to specify both row and column gaps at once.

Because `grid-column-gap` and `grid-row-gap` were not implemented in any browser at the time of writing—making it into Chrome Canary at the final moment—I have omitted them from my examples. For simple grids, these properties will serve to reduce the amount of CSS required.

LINE-BASED PLACEMENT AND THE SPAN KEYWORD

As I have shown, we don't need to use any kind of spanning properties to create a grid area spanning multiple grid lines. But the Grid Layout Module does include a `span` keyword, which can be used instead of specifying the end line explicitly.

The example shown in **FIG 1.3** could also be written like this:

```
.a {
  grid-column: 1 / span 3;
  grid-row: 1;
}

.b {
  grid-column: 5;
  grid-row: 1 / span 3;
}
```

```
.c {
  grid-column: 1;
  grid-row: 3;
}

.d {
  grid-column: 3;
  grid-row: 3;
}
```

Two new things show up here. The first is the `span` keyword; instead of positioning the div with a class of `.a` by saying, "Start at column line 1 and end at column line 4," I say, "Start at column line 1 and span 3 column lines." The result is the same.

I have also omitted the row or column end value where the content only spans to the next line because that is the default—no need to specify it.

LINE-BASED PLACEMENT WITH NAMED LINES

Keeping track of all of these line numbers soon gets old. Luckily, the Grid Layout Module provides a way to name lines, making it far easier to remember what goes where on the grid.

Let's continue with our example layout. When defining our grid, we can assign names to the lines, like so:

```
.wrapper {
  display: grid;
  grid-template-columns: [col1-start] 100px »
    [col1-end] 10px [col2-start] 100px [col2-end] »
    10px [col3-start] 100px [col3-end];
  grid-template-rows: [row1-start] auto [row1-end] »
    10px [row2-start] auto [row2-end];
}
```

Remember: *we are naming grid lines, not tracks.* In the value for `grid-template-columns` above, we name our first line `col1-start`. After that comes the 100-pixel track size. We name the line before the gutter `col1-end`, and then define the gutter track size of 10 pixels.
We can then position our items using those names, instead of numbers.

```
.a {
  grid-column: col1-start / col2-end;
  grid-row: row1-start;
}

.b {
  grid-column: col3-start;
  grid-row: row1-start / row2-end;
}

.c {
  grid-column: col1-start;
  grid-row: row2-start;
}

.d {
  grid-column: col2-start;
  grid-row: row2-start;
}
```

We will return to these named lines later in the book, when we look at a more complex use case. The nice thing about naming lines this way, though, is that once you've defined a grid, you can quickly arrange things without needing to think about their numerical position. For example, you can define `sidebar-start` and `sidebar-end` and know that any element positioned there will sit in the sidebar area.

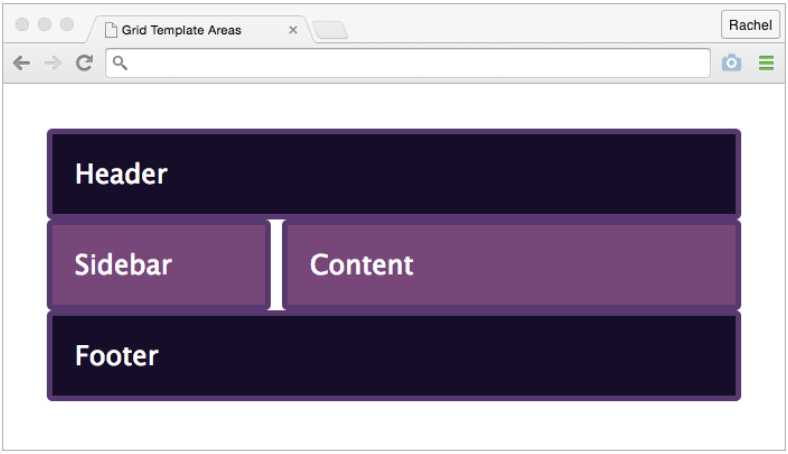

FIG 1.8: A simple layout using `grid-template-areas`.

GRID TEMPLATE AREAS

The final method of creating and positioning items on the grid is to use *grid template areas*. It's usually when I show people this method that they start to get almost as excited about Grid as I am. Here we create named grid areas, and then use the new property `grid-template-areas` to describe where on the grid these named areas sit.

This is my HTML: a small layout with a header, a sidebar, a content area, and a footer.

```
<div class="wrapper">
  <div class="box header">Header</div>
  <div class="box sidebar">Sidebar</div>
  <div class="box content">Content</div>
  <div class="box footer">Footer</div>
</div>
```

In my CSS, I have rules set up for each of the areas; I use the `grid-area` property to give them a name to refer to when defining the layout on the grid.

```
.sidebar { grid-area: sidebar; }
.content { grid-area: content; }
.header { grid-area: header; }
.footer { grid-area: footer; }
```

I now just need to declare my grid on the wrapper as before, but this time I also use `grid-template-areas` to define the layout using a kind of ASCII-art syntax.

```
.wrapper {
  display: grid;
  grid-template-columns: 200px 10px 200px 10px »
    200px;
  grid-template-rows: auto;
  grid-template-areas:
  "header header header header header"
  "sidebar . content content content"
  "footer footer footer footer footer"
}
```

Code example: http://bkaprt.com/cgl/01-06/

Repeating the name of an area causes the content to span those cells; one or more sequential period characters means that the cell remains empty. That's all you need to do to lay out a page using the CSS Grid Layout Module.

No clearing is required. I can add as much or as little content as I like into either the sidebar or the content area; the footer will always stay below both columns. The columns will also be the same height. No strange hacks required!

You do not have to add any non-semantic class names to your document; you can simply position the elements using the classes already applied to describe the content.

As I've suggested, Grid frees up source order, allowing us to organize our content in the most accessible and semantic way. Grid truly allows us to separate document structure from presentation. This means that we can easily redefine the layout in our media queries, and thus can create dramatically different

layouts for any viewport or particular use without compromising the document's structure.

And that's why I love the CSS Grid Layout Module. Read on to see more examples and discover what else you can do with the grid.

2

LAYING THINGS OUT ON THE GRID

NOW THAT WE'VE covered some of the basics of Grid, let's take a look at some common layouts and see how we might achieve them using this new method.

A THREE-COLUMN LAYOUT USING GRID-TEMPLATE-AREAS

To start, let's create a simple three-column layout.

Our HTML has the layout nested inside a wrapper `div` and includes a `header`; an `article` with a heading, a `div`, and an `aside` nested inside; an `aside`; and then a `footer`.

```
<div class="wrapper">
  <header class="mainheader">
    <h1>Excerpts from the book <em>The Bristol Royal
  Mail</em></h1>
  </header>

  <article class="content">
    <h1>Post letter boxes: position, violation,
    peculiar uses</h1>
    <div class="primary">
      <p>. . .</p>
    </div>
    <aside>
      <p>. . .</p>
    </aside>
  </article>

  <aside class="sidebar">

  </aside>

  <footer class="mainfooter">
    <p>. . .</p>
  </footer>
</div>
```

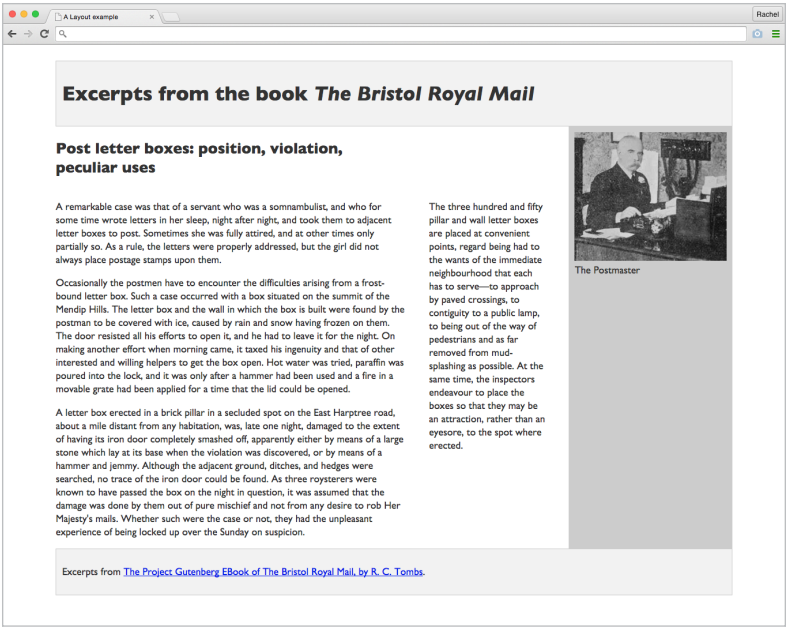

FIG 2.1: A simple three-column layout.

If we take a look at this in the browser, we can see the content displayed in document source order, since no positioning has yet been applied (**FIG 2.2**).

I want to use `grid-template-areas` to position my content in this example, so the first step is to define the main areas using the selectors that identify them.

```
.mainheader { grid-area: header; }
.content { grid-area: content; }
.sidebar { grid-area: sidebar; }
.mainfooter { grid-area: footer; }
```

I then create a grid on the `div` with a class of `.wrapper`. My grid has three columns: a 9-fraction-unit column, a 40-pixel gutter, and a 3-fraction-unit column. Because I've set

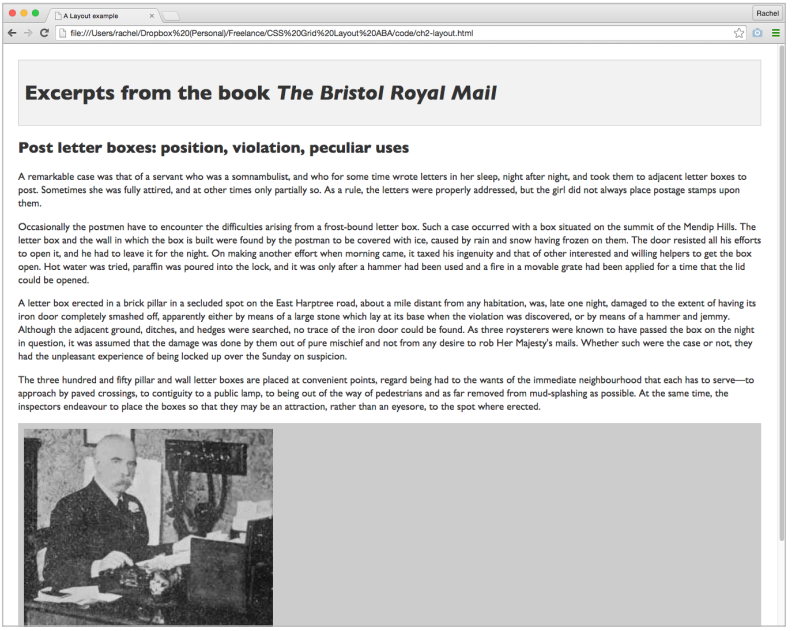

FIG 2.2: The layout before any positioning is added.

`grid-template-rows` to `auto`, we'll get as many rows as we need to accommodate our content.

Finally, I define the layout as the value of the `grid-template-areas` property. I repeat the `header` across all three columns of the first row. In the second row, I put the content in the left column and the sidebar on the right. The `footer` makes up the final row.

```
.wrapper {
  display: grid;
  width: 90%;
  margin: 0 auto 0 auto;
  grid-template-columns: 9fr 40px 3fr;
  grid-template-rows: auto;
  grid-template-areas:
    "header header header"
```

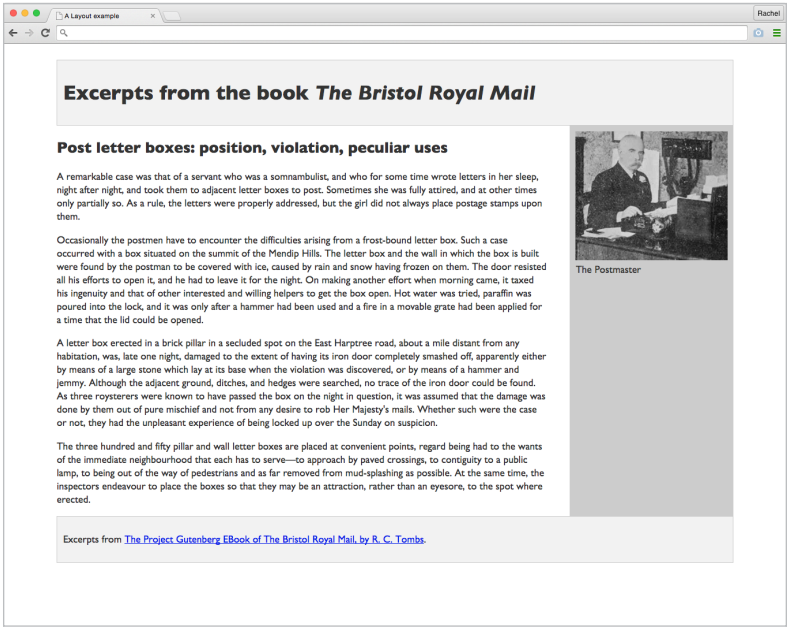

FIG 2.3: The layout after positioning the main content areas.

```
    "content . sidebar"
    "footer footer footer";
}
```

We now have a layout taking shape (**FIG 2.3**), all with just a few lines of CSS. That footer will stay put no matter which column is the longest. The background color on the sidebar extends right down to the footer.

Inside our `article` we have a heading, a `div` containing the primary content, and an `aside`. I'm going to also use Grid to position these, as it lets us create nested grids. I'll set up the grid areas of these nested items just as I did the main items.

```
.content .primary { grid-area: article-primary; }
.content aside { grid-area: article-secondary; }
.content > h1 { grid-area: chapterhead; }
```

I then create a new grid on `.content` and lay out our elements with the heading in the top left column, the primary content below it, and the `aside` to the right.

```
.content {
  display: grid;
  grid-template-columns: 9fr 40px 3fr;
  grid-template-rows: auto;
  grid-template-areas:
    "chapterhead . ."
    "article-primary . article-secondary";
  }
```

Code example: http://bkaprt.com/cgl/02-01/

That's all there is to it. We now have the layout shown at the beginning of this chapter (**FIG 2.1**).

Nested grids and subgrids

Our nested grid in this example is completely independent of the main grid. The container `.content` is positioned by the main grid, but the child elements are not—they acquire their grid from the way we set up `.content`.

This means we can't inherit column widths from the parent, which is a problem if you want to use flexible length units: you'll find it tricky to get the elements in the nested grid to line up with those in the outer grid.

The specification has a solution for this: the subgrid keyword. If we were to use `subgrid`, we would use it as a value for the `grid` property:

```
.content {
  grid: subgrid;
}
```

This comes in handy particularly when basing our layout on grid systems such as the twelve-column and sixteen-column grids that are currently popular. In most cases, you'll want the

children of nested elements to use the lines of the outer grid rather than implement their own grid.

At the time of writing, however, it looks as if the `subgrid` keyword will be moved to Level 2 of the specification, and there is currently no browser implementation of this functionality. My concern is that if this doesn't make it into the specification, authors will flatten their markup structure, removing semantic markup, in order to use Grid.

A boxy layout with line-based placement

Placement using template areas is straightforward and makes it very easy to position items into known page containers. To do more complex things, however—for example, to work with multiple-column grid systems—then line-based placement is the tool to use. My next example is an image layout, which could easily be a set of containers with content of any type (**FIG 2.4**).

The HTML for this example is very simple: a `div` with a class of `.wrapper` containing a `header` and our images.

```
<div class="wrapper">
  <header>
    <h1>Little boxes layout</h1>
  </header>

  <img src="images/balloon1.jpg" alt="Bristol
  Balloon Fiesta" class="box1">
  <img src="images/balloon2.jpg" alt="Bristol
  Balloon Fiesta" class="box2">
  <img src="images/balloon3.jpg" alt="Bristol
  Balloon Fiesta" class="box3">
  <img src="images/balloon4.jpg" alt="Bristol
  Balloon Fiesta" class="box4">
  <img src="images/balloon5.jpg" alt="Bristol
  Balloon Fiesta" class="box5">
  <img src="images/balloon6.jpg" alt="Bristol
  Balloon Fiesta" class="box6">
  <img src="images/balloon7.jpg" alt="Bristol
  Balloon Fiesta" class="box7">
```

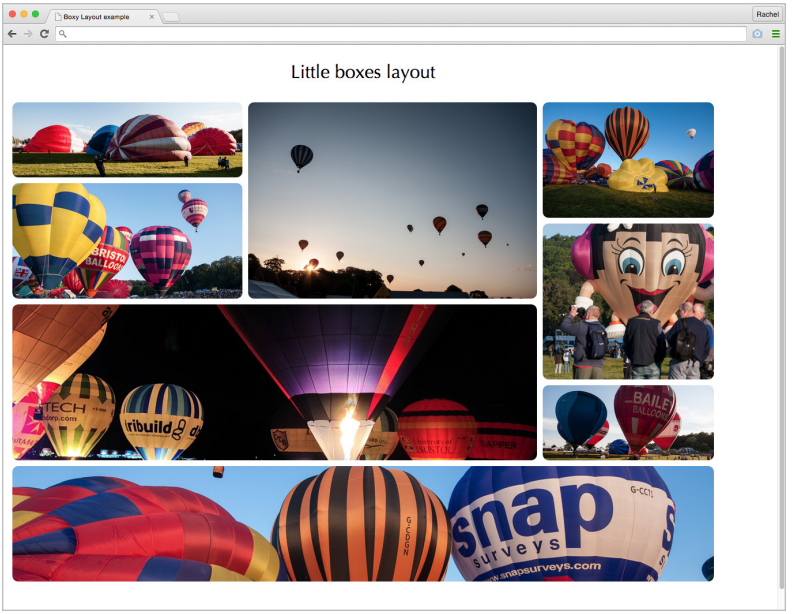

FIG 2.4: The completed boxy layout.

```
    <img src="images/balloon8.jpg" alt="Bristol
    Balloon Fiesta" class="box8">
  </div>
```

I declare a grid on the `.wrapper` element using the named grid lines syntax. I'm creating a grid containing thirteen column tracks, with a gutter track before each one. For the rows, I start with a row whose height value is `auto` for my heading. I then add fourteen row tracks, again with a 10-pixel gutter before each one.

```
.wrapper {
  display: grid;
  grid-template-columns:
    repeat(13, [gutter] 10px [col] 1fr);
  grid-template-rows:
    [row] auto repeat(14, [gutter] 10px [row] 60px);
}
```

The repeat keyword

Note that I used the `repeat` keyword when setting up this example. This simply saves me from having to write out my `gutter` and `col` pattern thirteen times. Between parentheses, I have the number of times that I want the pattern to repeat, followed by the pattern name.

The specification has recently been updated to allow the number of repeats to be `auto`. This means that a pattern can repeat as many times as the content requires. At the time of writing, this is not implemented in any browser. If it were, then our row pattern could be written as follows:

```
grid-template-rows:
    [row] auto repeat(auto, [gutter] 10px [row] »
      60px);
```

Positioning the boxes

With a grid defined, I can start to position the boxes. The `header` is going into the first row and will stretch across the layout.

```
header {
  grid-column: col / span gutter 12;
  grid-row: 1 / 1;
}
```

Using the named lines, we start at the first line named `col`, then span to gutter line 12.

I can place the individual images arbitrarily; as we know by now, Grid doesn't care about source order. The hardest thing about placing the elements is working out which line to place them on; browser tools for Grid would really help with this. It's not that difficult, though—it's just a matter of starting on the `col` and `row` line number you want the element to start on and spanning the required number of gutters.

```
.box1 {
  grid-column: col 1 / span gutter 4;
  grid-row: row 2 / span gutter 2;
}
.box2 {
  grid-column: col 1 / span gutter 9;
  grid-row: row 7 / span gutter 4;
}
.box3 {
  grid-column: col 1 / span gutter 12;
  grid-row: row 11 / span gutter 3;
}
.box4 {
  grid-column: col 10 / span gutter 3;
  grid-row: row 2 / span gutter 3;
}
.box5 {
  grid-column: col 5 / span gutter 5;
  grid-row: row 2 / span gutter 5;
}
.box6 {
  grid-column: col 1 / span gutter 4;
  grid-row: row 4 / span gutter 3;
}
.box7 {
  grid-column: col 10 / span gutter 3;
  grid-row: row 5 / span gutter 4;
}
.box8 {
  grid-column: col 10 / span gutter 3;
  grid-row: row 9 / span gutter 2;
}
```

Code example: http://bkaprt.com/cgl/02-02/

We now have the boxy grid layout. You can switch elements around by swapping their class or defining their grid position in the CSS without worrying about source order.

Layering items on the grid

If you are as old as I am, you may remember the days of using tables for layout. Grid is conceptually like using a table, but with a key difference: the table is not defined as a semantic table, but is defined in CSS. This means that it can be *redefined,* as we will see when we look at using Grid in responsive layouts.

Another important difference when using Grid Layout over table layout is that elements on the grid can overlap. If a grid area overlaps another grid area, you use `z-index` to control which element displays on top—just like with positioned elements. We can see how this works, and also demonstrate redefining an area, by adding a hover effect to our boxy layout.

First, I'll define a shadow and a z-index on `img:hover` to make sure that any image being hovered over will be displayed on top of the stack and also have a shadow.

```
img:hover {
  z-index: 10;
  box-shadow: 10px 10px 20px 0px rgba(0,0,0,0.75);
}
```

You should see the shadow when hovering over an image. I'm going to redefine the grid of each item on hover so the image grows and overlaps the neighboring images. For `.box1`, I will have two declarations: the first for the default state, the second for growing the box into the top and left gutters and spanning an additional gutter to the right and below.

```
.box1 {
  grid-column: col 1 / span gutter 4;
  grid-row: row 2 / span gutter 2;
}

.box1:hover {
  grid-column: gutter 1 / span gutter 5;
  grid-row: gutter 1 / span gutter 3;
}
```

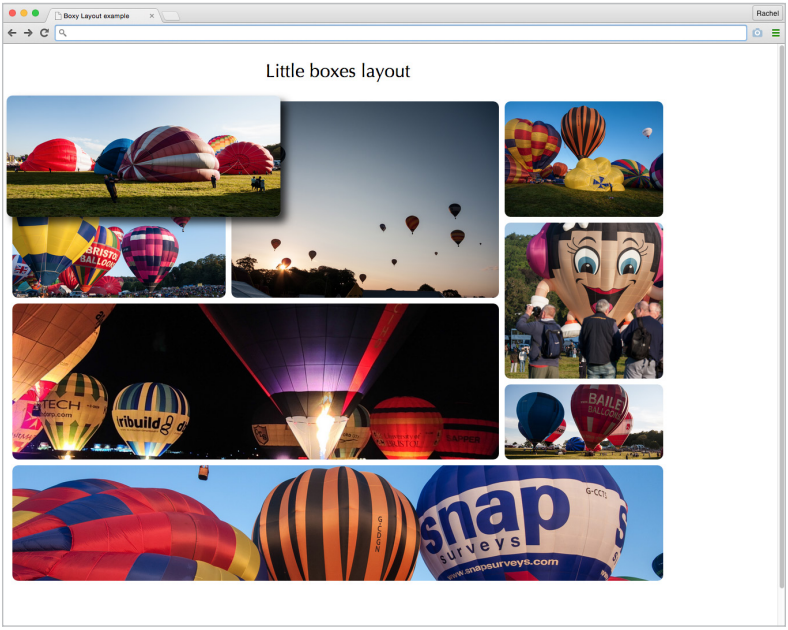

FIG 2.5: Redefining the grid on hover, showing how items can overlay one another.

I also need to increase the value for the number of repeats of my column pattern by 1 to 14, to allow the final item to have somewhere to grow.

```
grid-template-columns:
  repeat(14, (gutter) 10px (col) 1fr);
```

When I hover over the image, it now pops out and overlays its neighbors (**FIG 2.5**). You can do the same for the other images.

Code example: http://bkaprt.com/cgl/02-03/

You could, of course, create much nicer effects using animation to change the display of items on your grid. This probably

isn't an effect you'd use directly in production, but it demonstrates how you can change and layer items with Grid.

So far, we've gone over the two main ways of using the CSS Grid Layout Module, and have talked about some of the finer points and unimplemented elements of the specification. Next, we'll see how the Module will make it easier for us to control our layouts at multiple breakpoints.

3

CSS GRIDS AND RESPONSIVE DESIGN

IF YOU'VE MADE IT THIS FAR, you probably already have some idea of how useful CSS Grid Layout will be when working with responsive layouts. Proper separation of document source from presentation is something we have aimed for since the early days of CSS. Whenever I've been involved in developing a responsive design, though, I've always had to make compromises between source order and getting the most desirable layouts for each breakpoint. Grid, along with Flexbox, makes this far more straightforward.

To see how, let's look at some of the things we created in the last chapter, starting with the three-column layout using `grid-template-areas` (**FIG 2.1**).

Redefining grid-template-areas with media queries

Once again, I start by setting up all of my grid areas, for both the main grid and the nested grid on the `article`, with a class called `.content`.

```
.mainheader { grid-area: header; }
.content { grid-area: content; }
.sidebar { grid-area: sidebar; }
.mainfooter { grid-area: footer; }
.content .primary { grid-area: article-primary; }
.content aside { grid-area: article-secondary; }
.content > h1 { grid-area: chapterhead; }
```

At a breakpoint of 460 pixels, I start laying this out as a grid. For the main grid, I place the sidebar—which contains an image and is positioned last in the source—between the header and content.

I also declare a grid on `.content`, pulling the `aside` (defined as `article-secondary`) between the `chapterhead` and `article-primary` content (**FIG 3.1**).

```
@media only screen and (min-width: 460px) {
  .wrapper {
    display: grid;
    width: 90%;
    margin: 0 auto 0 auto;
```

FIG 3.1: The layout at 460 pixels.

```
    grid-template-columns: auto;
    grid-template-rows: auto;
    grid-template-areas:
      "header"
      "sidebar"
      "content"
      "footer";
  }
  .content {
    display: grid;
    grid-template-columns: auto;
    grid-template-rows: auto;
    grid-template-areas:
      "chapterhead"
      "article-secondary"
      "article-primary";
   }
   article aside { font-size: .75rem;}
}
```

Once we hit the 700-pixel breakpoint, we can go to two columns by redefining the grid on `.wrapper`. At this breakpoint, I'm going to leave the inner grid alone; three columns would be a little cramped.

```
@media only screen and (min-width: 700px) {
  .wrapper {
    grid-template-columns: 9fr 40px 3fr;
    grid-template-rows: auto;
    grid-template-areas:
      "header header header"
      "content . sidebar"
      "footer footer footer";
  }
}
```

Finally, I redefine the inner grid at 980 pixels, returning to the three-column layout I created in the last chapter.

```
@media only screen and (min-width: 980px) {
  .content {
    display: grid;
    grid-template-columns: 9fr 40px 3fr;
    grid-template-rows: auto;
    grid-template-areas:
      "chapterhead . ."
      "article-primary . article-secondary";
  }
  article aside { font-size: 100%;}
}
```

Code example: http://bkaprt.com/cgl/03-01/

Redefining the grid when using `grid-template-areas` is incredibly straightforward. It makes it much easier to tweak the layout at multiple breakpoints, allowing the content to determine what works best.

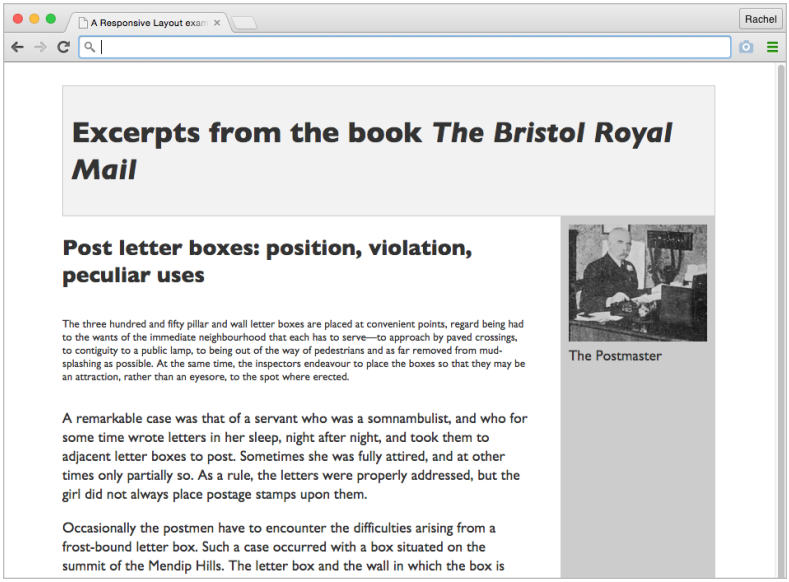

FIG 3.2: At two columns as we reach the 700-pixel breakpoint.

Redefining line-based placement

The second example we created in Chapter 2 was an image layout. A fair amount of screen real estate is necessary to view the layout in that format. We can add some media queries to adjust the layout to suit the viewport.

I've decided that unless the screen is wider than 720 pixels, I'll just allow the images to display one after the other; I've added a bottom margin to separate them. At 720 pixels, the fun starts. I'm going to define my multiple-column grid as before. Instead of the layout of differently sized boxes used for wider viewports, I'm just going to go for a two-column layout here. The top four boxes are the same height, spanning two gutters. I'll stagger the bottom images, making some span four gutters and others span two gutters.

```
@media only screen and (min-width: 720px) {
  .wrapper {
    display: grid;
    grid-template-columns:
      repeat(13, [gutter] 10px [col] 1fr);
    grid-template-rows:
      [row] auto repeat(13, [gutter] 10px [row] »
        60px);
  }
  header {
    grid-column: col / span gutter 12;
    grid-row: 1 / 1;
  }
  img {
    margin: 0;
  }
  .box1 {
    grid-column: col 1 / span gutter 6;
    grid-row: row 2 / span gutter 2;
  }
  .box2 {
    grid-column: col 7 / span gutter 6;
    grid-row: row 2 / span gutter 2;
  }
  .box3 {
    grid-column: col 1 / span gutter 6;
    grid-row: row 4 / span gutter 2;
  }
  .box4 {
    grid-column: col 7 / span gutter 6;
    grid-row: row 4 / span gutter 2;
  }
  .box5 {
    grid-column: col 1 / span gutter 6;
    grid-row: row 6 / span gutter 4;
  }
```

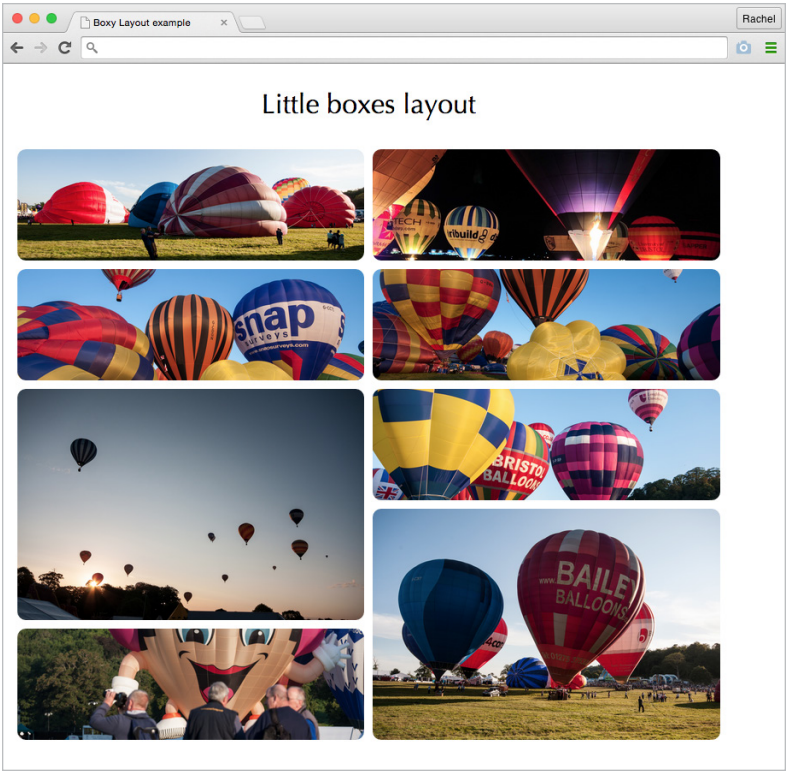

FIG 3.3: The two-column boxy layout.

```
.box6 {
  grid-column: col 7 / span gutter 6;
  grid-row: row 6 / span gutter 2;
}
.box7 {
  grid-column: col 1 / span gutter 6;
  grid-row: row 10 / span gutter 2;
}
```

```
  .box8 {
    grid-column: col 7 / span gutter 6;
    grid-row: row 8 / span gutter 4;
  }
}
```

Code example: http://bkaprt.com/cgl/03-02/

Let's return to my original layout at the 1200-pixel breakpoint. Note that I don't need to redefine my grid, just where elements sit on it. Because the grid is flexible, the tracks are wider or narrower depending on available viewport width.

Of course, you could also redefine your grid for different breakpoints, or use any combination of these methods in one layout. It's such a flexible tool!

Before we get too sad that we can't use this in production yet, it's worth noting that this is a fantastic way to prototype layouts because it makes it so easy to redefine areas. Even if you have to go back and rebuild your grid with some other method, you can use the Grid Layout Module and a supporting browser just to test out what works well for your content.

4

GRID, ANOTHER TOOL IN OUR KIT

THE CSS GRID LAYOUT MODULE will ultimately work alongside other layout methods to bring CSS layout up to date. Together with the Flexible Box Layout Module (Flexbox) and media queries, Grid Layout can help us build the types of sites and layouts we need to build today.

People new to Grid Layout commonly wonder how Grid works with Flexbox. When should we choose to use one or the other (in an ideal world where both have support)? On the www-style mailing list, Tab Atkins gave an excellent answer to this question:

> *Flexbox is for one-dimensional layouts—anything that needs to be laid out in a straight line (or in a broken line, which would be a single straight line if they were joined back together).*
>
> *Grid is for two-dimensional layouts. It can be used as a low-powered flexbox substitute (we're trying to make sure that a single-column/row grid acts very similar to a flexbox), but that's not using its full power.*
>
> *Flexbox is appropriate for many layouts, and a lot of "page component" elements, as most of them are fundamentally linear. Grid is appropriate for overall page layout, and for complicated page components which aren't linear in their design.*
>
> *The two can be composed arbitrarily, so once they're both widely supported, I believe most pages will be composed of an outer grid for the overall layout, a mix of nested flexboxes and grid for the components of the page, and finally block/inline/table layout at the "leaves" of the page, where the text and content live.*

—TAB ATKINS, MAY 6, 2013 (http://bkaprt.com/cgl/04-01/)

We can take a look at how this might work by creating the layout shown in FIG 4.1. It's a fairly standard layout: navigation, followed by a large feature image and a main `article`, then a set of boxes featuring content of different heights, and, finally, a `footer`.

The HTML for the body of my page looks like this:

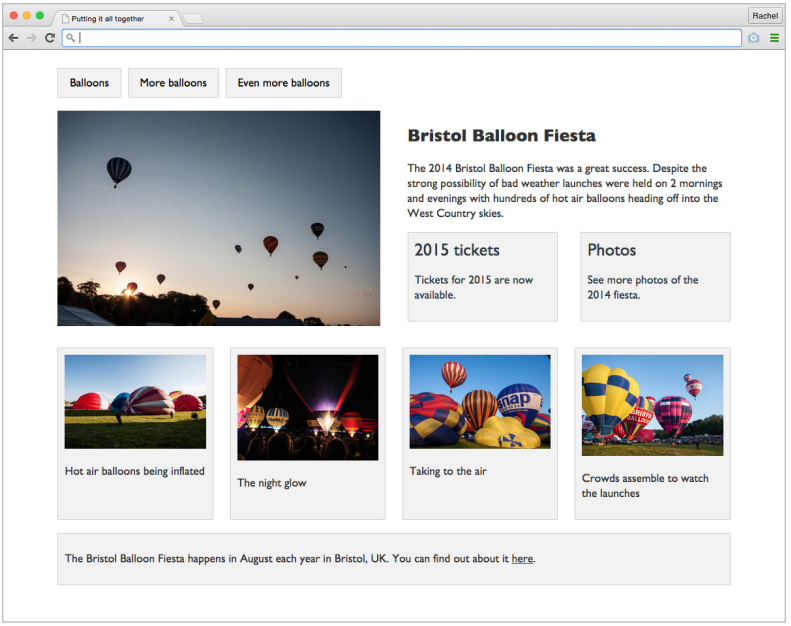

FIG 4.1: A layout combining Grid Layout with other methods.

```
<div class="wrapper">
  <header class="mainheader">
    <nav>
      <ul>
        <li><a href="">Balloons</a></li>
        <li><a href="">More balloons</a></li>
        <li><a href="">Even more balloons</a></li>
      </ul>
    </nav>
  </header>

  <aside class="feature-pull"><img src="images/
    balloon5.jpg" alt="Bristol Balloon Fiesta">
  </aside>
```

```
    <article class="feature">
      <h1>Bristol Balloon Fiesta</h1>
      <p>...</p>

      <ul class="cta-list">
        <li class="box"><li>
        <li class="box"><li>
      </ul>
    </article>

    <ul class="gallery">
      <li class="box"><img src="images/balloon1.jpg"
    alt="Inflating balloons">
      <p>Hot air balloons being inflated</p></li>
      <li class="box"></li>
      <li class="box"></li>
      <li class="box"></li>
    </ul>
    <footer class="mainfooter box"></footer>
  </div>
```

I've added a class called `.box` to the elements to give them a basic box style so we can clearly see the layout; I've also written some CSS for basic styling. This gives us the linearized view shown in **FIG 4.2**.

At a 460-pixel breakpoint, I start using Grid Layout to position the main content areas on the page. Inside the media query, I set up those main areas and then create a linearized layout to arrange them in the order I want.

```
@media only screen and (min-width: 460px) {

  .mainheader { grid-area: header; margin: 0 0 »
    20px 0;}
  .feature-pull { grid-area: featurepull; }
  .feature { grid-area: feature; }
  .gallery { grid-area: secondary; }
  .mainfooter { grid-area: footer; }
```

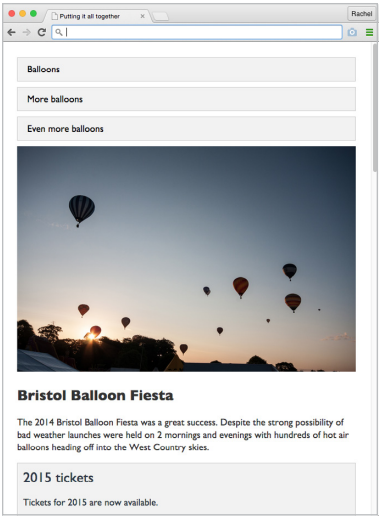

FIG 4.2: The layout prior to adding CSS to position elements.

```
  .wrapper {
    display: grid;
    width: 90%;
    margin: 0 auto 0 auto;
    grid-template-columns: auto;
    grid-template-rows: auto;
    grid-template-areas:
      "header"
      "feature"
      "featurepull"
      "secondary"
      "footer";
  }
}
```

At 760 pixels, I move to a two-column layout with a gutter. This allows me to place the feature image to the left of the `article`.

```
@media only screen and (min-width: 760px) {
  .wrapper {
    grid-template-columns: 48% 4% 48%;
    grid-template-rows: auto;
    grid-template-areas:
      "header header header"
      "featurepull . feature"
      "secondary secondary secondary"
      "footer footer footer";
}
```

FIG 4.3 shows a number of elements on the page that haven't yet been laid out to match the initial design (**FIG 4.1**). The navigation runs right across the page, the small images need to be formatted into a row of four, and two calls to action that should be side by side are stacked under the main content.

It would be possible to achieve this with Grid, but Flexbox is probably a better fit for such small interface elements. Flexbox has the ability to wrap rows, plus other features that are useful when dealing with the smaller parts of a user interface.

Inside the media query for the first breakpoint, I set the `.mainheader ul` and `.gallery` selectors to `display: flex`. I add to that at the second breakpoint by turning the wrapper for the two call-to-action boxes (`.cta-list`) into a Flex container.

My simple layout is now complete and resembles what is displayed in **FIG 4.1**.

Code example: http://bkaprt.com/cgl/04-02/
& http://bkaprt.com/cgl/04-03/

One thing I appreciate about using Grid and Flexbox like this is how infrequently we need to add wrapper divs purely to create the layout. Most of the existing layout methods currently require redundant markup. It's nice to know I'm not adding bulk to my pages just for CSS purposes.

The method outlined in this chapter provides insight into how we will be working with CSS layout in the not-so-distant future. In my final chapter, I'll talk about that future and how we can welcome it.

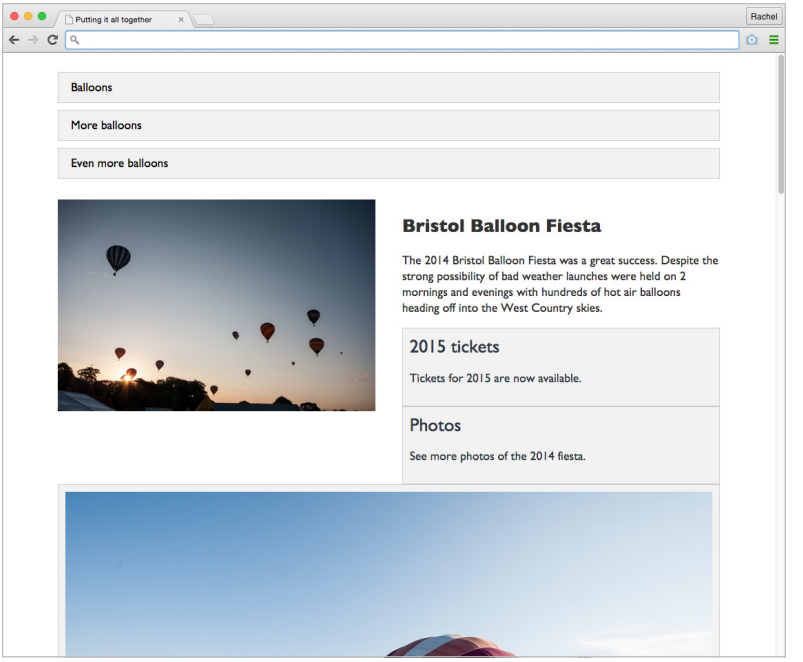

FIG 4.3: The layout after positioning the main page areas.

5

WHAT'S NEXT FOR GRID?

I HOPE that this book has given you a good introduction to CSS Grid Layout and encouraged you to explore further. In this concluding chapter, I'll summarize the current state of browser implementations and offer some thoughts about why it is worth exploring the specification even if we can't yet use it in production.

CSS GRID LAYOUT IN BROWSERS

As I write this book, all signs suggest that 2015 will be the year of CSS Grid Layout. By the end of this year, we very well may see an implementation of Grid in all major browsers; I'll keep an up-to-date list of browser support information at Grid by Example (http://bkaprt.com/cgl/05-01/).

Blink

All of the examples in this book work right now unprefixed in Chrome with the Experimental Web Platform Features flag enabled. Chrome, along with Opera and other browsers, uses the Blink rendering engine. The Blink implementation work is being sponsored by Bloomberg and carried out by open source consultancy Igalia (http://bkaprt.com/cgl/05-02/).

Igalia plans to send the "Intent to Ship" to the Blink mailing list this year (http://bkaprt.com/cgl/05-03/). Once Grid is out from behind a flag, there will be greater potential to use it for applications, especially those designed for mobile (where Chrome is the majority browser).

Webkit

Along with the Blink implementation, Igalia has also been working on the one for Webkit. It's currently available in Webkit Nightly Builds (http://bkaprt.com/cgl/05-04/), with `the -webkit` prefix. I hope this means that we will see Grid in Safari in the future.

Mozilla

Mozilla announced an "Intent to Implement" CSS Grid Layout on February 2, 2015 (http://bkaprt.com/cgl/05-05/). In October 2015, many of the examples I developed began to work in Firefox Nightly builds when enabling the flag `layout.css.grid.enabled`. I'm excited to see that implementation take shape.

Internet Explorer

At the time of writing, Internet Explorer and Edge support the old implementation with the `-ms` prefix. My 2012 *24 Ways* article demonstrates how this implementation works (http://bkaprt.com/cgl/05-06/). Microsoft has publicly indicated that updating Grid to the new specification is high priority on the Edge backlog (http://bkaprt.com/cgl/05-07/).

POLYFILLING GRID

François Remy has developed a polyfill that implements much of the Level 1 syntax (http://bkaprt.com/cgl/05-08/).

Once there is a good level of browser support, I would recommend using Grid and offering a fallback where Grid is not supported. That fallback might involve a polyfill, but could also be a simpler layout.

WHY LOOK AT CSS GRID LAYOUT NOW?

Moving to a new layout method in CSS is hard. Adopting modern layout methods using Grid and Flexbox feels just as difficult as making the leap away from using tables back in the early days of CSS. But as browser support develops over the next several months, I believe there will be opportunities for some of us to

start using Grid Layout in production. With Grid support in Blink and potentially in Safari if the Webkit implementation is adopted, there will be use cases for mobile devices; in the case of no Grid support, a linearized layout would be acceptable and the so-called "desktop" version of the site could use older layout methods.

Using Grid for prototyping content-driven responsive designs

CSS Grid Layout is currently a great tool for wireframing and prototyping responsive design. Once it is not behind a flag in Chrome, those prototypes could easily be shared with clients. I'm a great fan of letting content, rather than specific devices, dictate breakpoints. Grid Layout makes it simple to play around with the breakpoints and how content is reorganized within them, even if ultimately you need to use older methods to build your final layouts.

Offering feedback on the specification

One of the most important reasons for designers and developers to start experimenting with CSS Grid Layout and other emerging specifications is so that we can comment on (and help shape) the developing standards.

Anyone can join the www-style mailing list, set up a filter for "css-grid," and follow along with the discussions. Many of those discussions are heavily technical, but sometimes feedback is requested on terminology, or whether developers actually use a certain feature or not. This is crucial: developers so often complain that a specification does not meet their needs, and yet they rarely get involved at a point when changes could be addressed. Specification writers are usually not practicing web developers, but the documents they work on are available as Editor's and Working Drafts. The process is open for our feedback if we are willing to give it.

Take a look at the resources I've listed here, experiment with the examples in this book, and get involved with offering

feedback on the specification or in logging bugs against browser implementations. I believe that CSS Grid Layout is the layout method we have been waiting for, and I hope that we'll be using it in production very soon.

RESOURCES

Find out more about CSS Grid Layout and stay current with browser support and developments in the specification. The latest version of the specification: http://bkaprt.com/cgl/06-01/.

Search "css-grid" on the www-style mailing list: http://bkaprt.com/cgl/06-02/ to read and participate in current discussion on the Grid spec.

Grid by Example, my resource site on CSS Grid Layout: http://bkaprt.com/cgl/05-01/ is a collection of step-by-step examples and experiments with Grid. I'll add to this resource frequently as the spec and browser support develop.

Also see these examples from Igalia, which is doing the Blink and Webkit implementation of Grid: http://bkaprt.com/cgl/06-03/.

ACKNOWLEDGEMENTS

I owe a debt to many people who have spent time discussing Grid with me, pointing out my mistakes, and being interested in my opinions. In particular, I'm grateful to Spec editors Tab Atkins, Jr. and fantasai, and the developers at Igalia. Also, I appreciate every conference audience member who has come up and chatted with me after my Grid presentations. My ability to understand and teach Grid is informed by these conversations—revealing which things are confusing, and which are truly exciting for other developers and designers. Thank you.

REFERENCES

Shortened URLs are numbered sequentially; the related long URLs are listed below for reference.

Introduction

00-01 http://www.w3.org/TR/2011/WD-css3-grid-layout-20110407/

Chapter 1

01-01 https://github.com/abookapart/css-grid-layout-code/blob/master/ch1-line-based.html

01-02 https://github.com/abookapart/css-grid-layout-code/

01-03 https://github.com/abookapart/css-grid-layout-code/blob/master/ch1-line-based-shorthand.html

01-04 https://github.com/abookapart/css-grid-layout-code/blob/master/ch1-line-based-grid-area.html

01-05 https://github.com/abookapart/css-grid-layout-code/blob/master/ch1-line-based-span.html

01-06 https://github.com/abookapart/css-grid-layout-code/blob/master/ch1-line-based-grid-area.html

Chapter 2

02-01 https://github.com/abookapart/css-grid-layout-code/blob/master/ch2-layout.html

02-02 https://github.com/abookapart/css-grid-layout-code/blob/master/ch2-boxy.html

02-03 https://github.com/abookapart/css-grid-layout-code/blob/master/ch2-boxy-hover.html

Chapter 3

03-01 https://github.com/abookapart/css-grid-layout-code/blob/master/ch3-layout.html

03-02 https://github.com/abookapart/css-grid-layout-code/blob/master/ch3-boxy.html

Chapter 4

04-01 http://lists.w3.org/Archives/Public/www-style/2013May/0114.html

04-02 https://github.com/abookapart/css-grid-layout-code/blob/master/ch4-layout.html

04-03 https://github.com/abookapart/css-grid-layout-code/blob/master/ch4-layout-styles.css

Chapter 5

05-01 http://gridbyexample.com/

05-02 http://www.igalia.com/

05-03 http://blogs.igalia.com/mrego/2015/01/08/css-grid-layout-2014-recap-implementation-status/

05-04 http://nightly.webkit.org/

05-05 http://article.gmane.org/gmane.comp.mozilla.devel.platform/12343

05-06 http://24ways.org/2012/css3-grid-layout/

05-07 https://wpdev.uservoice.com/forums/257854-microsoft-edge-developer/suggestions/6514853-update-css-grid

05-08 http://fremycompany.com/BG/2014/CSS-Grid-Polyfill-Level-1-346/

Resources

06-01 http://www.w3.org/TR/css-grid-1/

06-02 https://www.w3.org/Search/Mail/Public/search?type-index=www-style&index-type=t&keywords=%5bcss-grid%5d&search=Search

06-03 http://igalia.github.io/css-grid-layout/index.html

INDEX

ABOUT A BOOK APART

We cover the emerging and essential topics in web design and development with style, clarity, and above all, brevity—because working designer-developers can't afford to waste time.

COLOPHON

The text is set in FF Yoga and its companion, FF Yoga Sans, both by Xavier Dupré. Headlines and cover are set in Titling Gothic by David Berlow.

ABOUT THE AUTHOR

Rachel Andrew lives in Bristol, England. She is the cofounder of edgeofmyseat.com, the web development company behind Perch CMS. She works on everything from product development to DevOps to CSS, and writes about these subjects on her blog at rachelandrew.co.uk.

Rachel has been working on the web since 1996 and writing about the web for almost as long. She's written several books including the bestselling *CSS Anthology* from Sitepoint, and recent ventures into self-publishing have produced *The Profitable Side Project Handbook* and *CSS3 Layout Modules, Second Edition*. She is a regular columnist for *A List Apart* as well as other publications online and in print. When she's not writing, Rachel often works with other authors as a technical editor.

Rachel is a keen distance runner who encourages people to join her for a run when attending conferences, with varying degrees of success! You can find her on Twitter as @rachelandrew.