1

SECOND EDITION

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
Managing Editor: Lisa Maria Martin
Editor: Caren Litherland
Copyeditor: Katel LeDû
Proofreader: Katel LeDû
Book Producer: Ron Bilodeau

Editor, first edition: Caren Litherland
Technical Editor, first edition: Paul Lloyd
Copyeditor, first edition: Lisa Maria Martin
Compositor, first edition: Rob Weychert
Ebook Producer, first edition: Ron Bilodeau

ISBN: 978-1-937557-91-1

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

TABLE OF CONTENTS

FOREWORD

WHAT DOES IT LOOK LIKE, when a new web feature is tested for years, honed to a fine edge, and launched in multiple browsers almost simultaneously, catapulting its global support from nothing to well over eighty percent in the space of a few weeks?

It looks like CSS Grid.

For those of us who have watched web standards develop for lo these many years, what happened with Grid was almost incomprehensible. We're used to watching one browser pick up a new standard, and then wait years for the others to join the fun. We're used to seeing these slowly emerging implementations riddled with gaps, or having to change defined behaviors midstream because flaws in the specification were uncovered long after shipping. Flexbox suffered this.

But Grid—no, Grid arrived in a fusillade of browser updates, with robust consistency and a small rump of glitches and oddities that were quickly smoothed out. The ship that Internet Explorer (yes!) launched in 2012 set sail as an armada in the spring of 2017.

That was then. What about now?

Now we have two years of slowly growing experience with Grid. Sites have shipped using it for layout almost unheralded, because there was no need for clever hackery to make it work. It does what it claims to do, what it was *designed* to do, with efficiency and elegance.

And now, with that experience behind us, the specification is being updated to address some of the rare limitations that existed in the first version of Grid. That's why you're lucky to have this book in front of you. No one is better qualified than Rachel Andrew to explain the basics *and* the evolution of Grid. Whether this is your first foray into Grid or a refresher course on a technology you already rely on, you'll find what you need here. And, quite probably, you'll find nearly everything you might *want* in a layout language. Savor it. We may not see its like again.

—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. We live in a very different world from the one in which I learned my craft!

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. This early version of the specification appeared in Internet Explorers 10 and 11 and was adopted by the W3C. Updated versions of the specification ultimately shipped in Chrome, Firefox, and Safari within weeks of one another in March 2017. Microsoft Edge updated from the old Internet Explorer specification in October of that year.

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 a wider variety of 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.

The first edition of this book was written in 2015, before Level 1 of the specification shipped in browsers. As I write this update four years later, Level 2 of the specification is being implemented in Firefox, and Grid Layout support hovers around eighty-nine percent globally[1], according to CanIUse.com. The second edition, like the first, is an ode to the elegance and power of the specification. And now that Grid has shipped in browsers, I hope this book can also serve as your guide as you start using Grid Layout in your projects today.

1

WHAT IS CSS GRID LAYOUT?

THE CSS GRID LAYOUT MODULE 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 a flexible or fixed layout grid. The grid can be redefined using media queries. This makes CSS Grid Layout an incredibly powerful tool—one that the web has been waiting for ever since we began doing layout with CSS instead of tables.

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 through a series of examples. The example code is linked so you can use it as a starting point for your own explorations. Everything described—with the exception of the final section on Grid Layout Level 2—is supported in Chrome, Firefox, Safari, and Edge, plus the myriad Chromium-based browsers available today.

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`, making this element our *grid container*.

```
.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 100px 100px;
  grid-template-rows: 100px 100px;
}
```

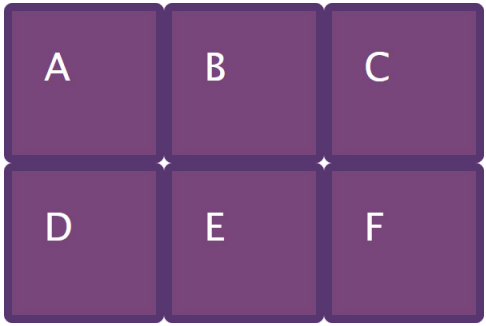
Code example: http://bkaprt.com/cgl-2/01-01/

You can find this, and all of the examples in this book, on GitHub. I'll reference the file below each example, like I've done above.

Here, I've created a grid with three 100-pixel-wide columns, and two 100-pixel-tall rows.

If we take a look at our page after we've declared a grid, we'll see that the child elements have placed themselves on the grid (**FIG 1.1**). They do this according to Grid's auto-placement rules, which simply fill each cell in turn with a direct child of the grid container.

FIG 1.1:
Grid automatically fills each consecutive cell with a direct child of the grid container.



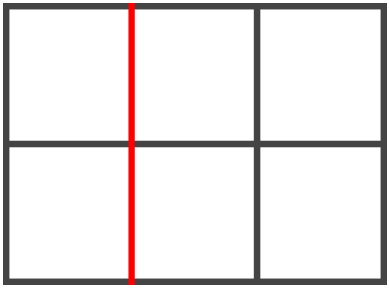


FIG 1.2: The highlighted grid line is column line 2.

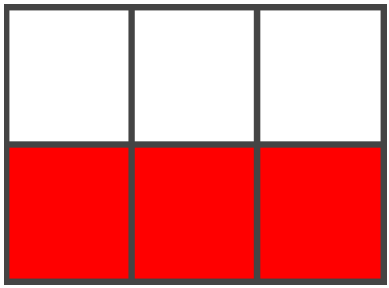


FIG 1.3: Here, I've highlighted the track between row lines 2 and 3.

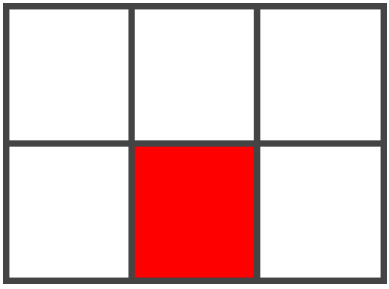


FIG 1.4: The highlighted grid cell in this image is between row lines 2 and 3 and column lines 2 and 3.

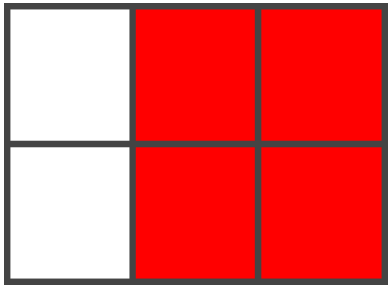


FIG 1.5: The highlighted grid area in this image falls between row lines 1 and 3 and column lines 2 and 4.

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, but they can also be named (**FIG 1.2**).

Grid track

A *grid track* is the space between two grid lines. It can be either horizontal or vertical (**FIG 1.3**).

Grid cell

A *grid cell*—the space between four grid lines—is the smallest possible unit on the grid (**FIG 1.4**). Conceptually, it is exactly like a table cell.

Grid area

A *grid area* is any area of the grid bound by four grid lines (**FIG 1.5**). It can contain a number of grid cells.

To see a visual representation of your grid as you work on it, you can use the Grid Inspector included with Firefox Developer Tools. Select your Grid Container and then look at the Layout Panel, where you can turn on the grid lines, see line numbers, and highlight grid cells (**FIG 1.6**).

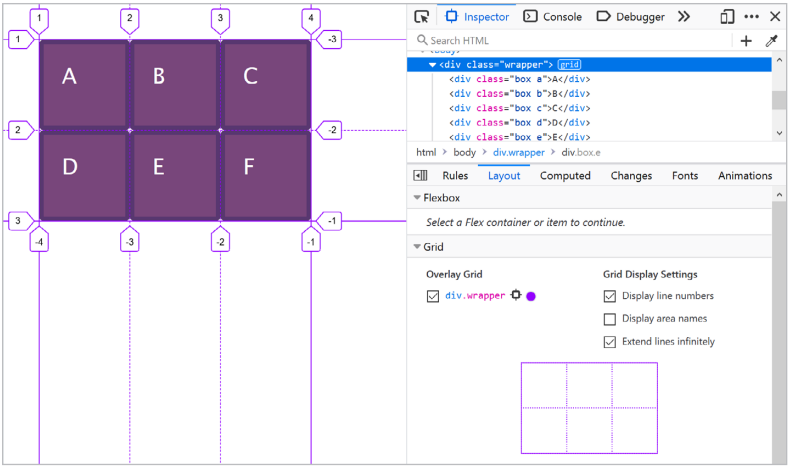


FIG 1.6: Inspecting our grid with the Firefox Grid Inspector highlights many of the parts of a grid layout.

The explicit and implicit grid

In our initial example, we created an *explicit* grid: we declared `grid-template-columns` and `grid-template-rows`, and the child elements of our grid container automatically slotted into the cells created by those grid tracks.

If we don't create enough cells, or place something outside of the explicit grid, Grid creates *implicit* grid tracks for us. This means that we can remove our `grid-template-rows` property, and our items will still place themselves on the grid. The rows are no longer 100 pixels tall; instead, they are auto-sized. It's safe to assume that an auto-sized track will be large enough to fit the content.

More ways to define your grid

When using Grid Layout, we do a lot of work on the grid container, setting up our grid so that it's ready for us to place items into it. But before we move on to placing items, let's have a look at some of the additional ways we can define our grid.

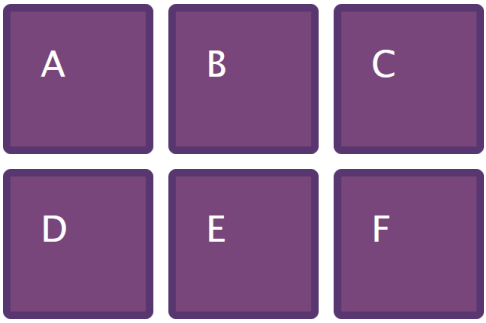


FIG 1.7: The `gap` property creates gaps between our tracks. It's shorthand for `column-gap` and `row-gap`.

Gaps between grid tracks

Sometimes we might want to create gaps between our items. We can do this by using the `gap` property, or individual properties of `column-gap` and `row-gap`.

```
.wrapper {
  display: grid;
  grid-template-columns: 100px 100px 100px;
  grid-template-rows: 200px 200px;
  gap: 10px;
}
```

Code example: http://bkaprt.com/cgl-2/01-03

Sizing implicit rows

When describing the implicit grid, I explained that the tracks created in it are auto-sized. This is the initial value of such tracks—but we can specify a size for them using the `grid-auto-rows` and `grid-auto-columns` properties.

In the following example, I've used `grid-auto-rows` with a value of `100px` to create rows in the implicit grid that are 100 pixels tall, rather than specifying explicit tracks.

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

Code example: http://bkaprt.com/cgl-2/01-04

The `minmax()` function

If we create rows that have a fixed height, as in the previous example, and then add more content to our items, we'll probably run into a situation where the content overflows the fixed-height track (**FIG 1.8**).

Designs that appear to have fixed-size rows but that can accommodate extra content are a nice feature of Grid Layout. We can achieve this flexibility by using the `minmax()` function for our track sizing.

The `minmax()` function allows us to pass in a minimum and a maximum value. In the following example, I've passed in a minimum of `100px` and a maximum of `auto`. This means that if there is less content in the track than would make it 100 pixels tall, it will nevertheless be at least 100 pixels tall. The maximum of `auto` means that the track can get as tall as necessary to fit the content. You can use `minmax()` anywhere you can add a fixed track size.

```
.wrapper {
  display: grid;
  grid-template-columns: 100px 100px 100px;
  grid-auto-rows:minmax(100px, auto);
  gap: 10px;
}
```

Code example: http://bkaprt.com/cgl-2/01-05

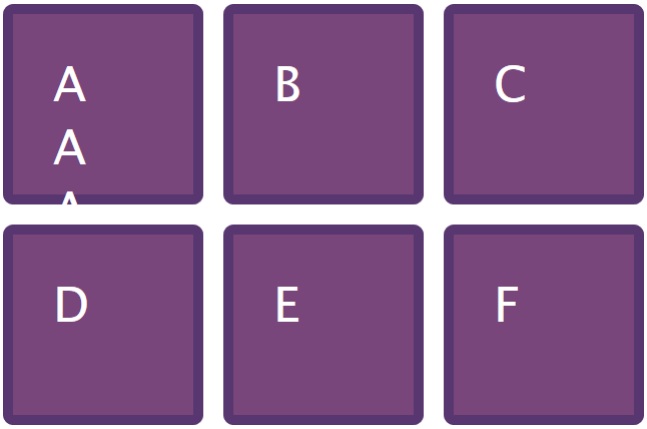


FIG 1.8: Content overflows if it exceeds the fixed-height track.

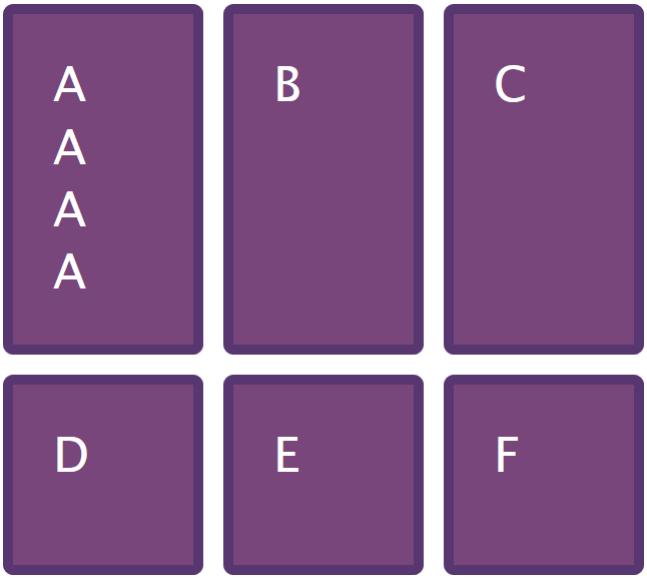


FIG 1.9: By using `minmax()`, we can make the track expand as necessary.

The fr unit

You can create your tracks with any valid CSS length unit, or with percentages. But you can also size tracks using the Grid-specific `fr` unit. This value represents a fraction of the available space in the grid container. If we change our example to use `fr` units, we can see how it works:

```
.wrapper {
    width: 600px;
  display: grid;
  grid-template-columns: 2fr 1fr 1fr;
    grid-auto-rows:minmax(100px, auto);
  gap: 10px;
}
```

I've made my first track `2fr`, and the second and third tracks `1fr` each. This means that the available space in the grid container is divided into four: two parts are assigned to the first track, and one part each to the second and third tracks (**FIG 1.10**).

Note that we're not distributing *all* of the available space in the grid container, only the space left over after laying out the content. A larger amount of content in the third track means less remaining space to assign to the others (**FIG 1.11**).

Now that you know about `minmax()`,it might be useful to understand that `1fr` is really `minmax(auto, 1fr)`. In other words, Grid assigns `auto` (enough space for the content) and then parcels out the leftover space. If we want to force even space distribution, we can use `minmax(0, 1fr)` explicitly. This treats each track as if it has a size of `0`, and shares out all of the space the items need to fit into.

```
.wrapper {
  display: grid;
  grid-template-columns: minmax(0,2fr) minmax(0,1fr)
  minmax(0,1fr);
    grid-auto-rows:minmax(100px, auto);
  gap: 10px;
}
```

Code example: http://bkaprt.com/cgl-2/01-06

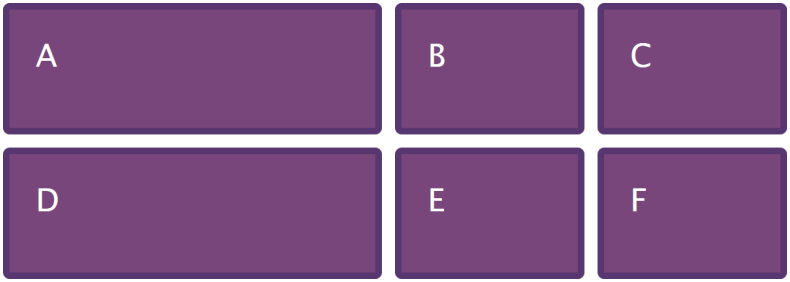


FIG 1.10: Using the `fr` unit to create flexible tracks.

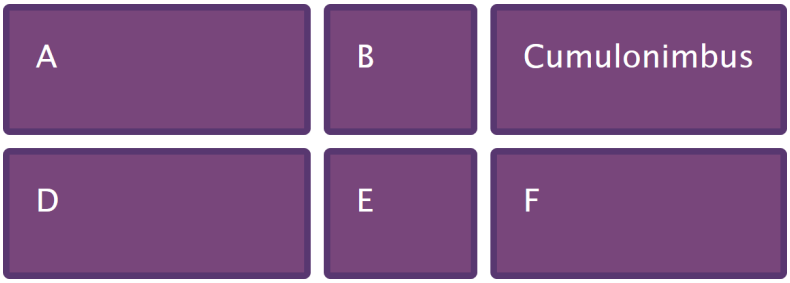


FIG 1.11: Something that takes up a lot of room in one track means less space is left over for other tracks.

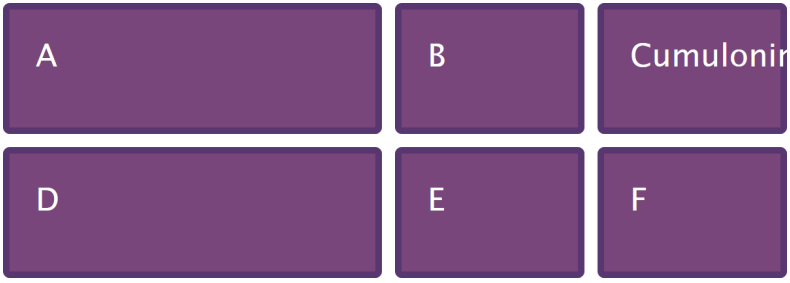


FIG 1.12: We can use `minmax(0,1fr)` to force an even distribution of all space, not just leftover space.

Using repeat() notation

If we're creating a grid with a large number of tracks of equal size, we can repeat all, or a section of, our track listing. For example, say we want to create a twelve-column grid using `minmax(0,1fr)` to make every column the same size. To do this, we could use the following code:

```
.wrapper {
  display: grid;
  grid-template-columns: minmax(0,1fr) minmax(0,1fr)
  minmax(0,1fr) minmax(0,1fr) minmax(0,1fr)
  minmax(0,1fr) minmax(0,1fr) minmax(0,1fr)
  minmax(0,1fr) minmax(0,1fr) minmax(0,1fr)
  minmax(0,1fr);
}
```

Or, we could dramatically simplify our CSS with `repeat()`:

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(12, minmax(0,1fr));
}
```

Code example: http://bkaprt.com/cgl-2/01-07

When we use `repeat()`, we place comma-separated values between parentheses. The value before the comma stands for the number of times a pattern should repeat; the value after the comma refers to the pattern. We can repeat a single track value or a track listing.

Using the auto-fill and auto-fit keywords

We can combine all of the things we've learned so far to create a very useful pattern. Let's say we want to have as many tracks as will fit into our grid container, and we want the tracks to have a minimum size. This enables a responsive number of column tracks without relying on media queries to add breakpoints.

The following example creates a repeating pattern of tracks that are exactly two hundred pixels wide. If we open the code example in a browser and resize the window, we'll see that additional grid tracks are created or removed depending on how much space is available, and the auto-placed items are reflowed. This is achieved by using the `auto-fill` keyword instead of a number before the comma in our repeat notation.

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(auto-fill, 200px);
}
```

Code example: http://bkaprt.com/cgl-2/01-08

This isn't quite what we want, though—these automatically created tracks are always two hundred pixels wide. So unless our grid container can be neatly divided by two hundred, we'll have a gap at the end.

We can get the result we're after by combining `repeat`, `auto-fill`, and `minmax()`:

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(auto-fill,
  minmax(200px, 1fr));
}
```

Code example: http://bkaprt.com/cgl-2/01-09

The `minmax()` function has a minimum value of `200px` (our desired minimum). It has a maximum value of `1fr`, which distributes any leftover space across the tracks that have been created. Thus we get as many flexible tracks as will fit into our container, with a minimum size of two hundred pixels.

Placing items by line number

All of the work we've done so far has been on the grid container. Our items have been auto-placed into cells created by our tracks. Now let's turn to the grid items themselves, and see how they can be placed on the grid.

The simplest method to use is line-based placement. We can use the following rules to place an element whose class is `.a` so that it spans two column tracks and two row tracks, from line 1 to line 3 for both columns and rows:

```
.a {
  grid-row-start: 1;
  grid-column-start: 1;
  grid-row-end: 3;
  grid-column-end: 3;
}
```

Code example: http://bkaprt.com/cgl-2/01-10

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-row: 1 / 3;
  grid-column: 1 / 3;
}
```

Code example: http://bkaprt.com/cgl-2/01-11

By drawing a box around the area we want our content to go into, line-based placement creates a *grid area*. An even shorter shorthand than the `grid-column` and `grid-row` properties is the `grid-area` property. Here is the order of values:

1. `grid-row-start`
2. `grid-column-start`
3. `grid-row-end`
4. `grid-column-end`

This gives us:

```
.a {
  grid-area: 1 / 1 / 3 / 3;
}
```

Code example: http://bkaprt.com/cgl-2/01-12

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

A grid area can span as many individual grid cells as required. We just need to specify the line where the content will start and the line where it will end. In the code example, I've placed four items onto the grid using line-based positioning (**FIG 1.13**).

The Firefox Grid Inspector makes it easy to place items because it indicates the line numbers (**FIG 1.14**).

Note that the source order of these child elements doesn't affect where we can place each one. Having said that, though, it's not a good idea to make the visual display of items diverge from the order of items in the source. The source determines the order that screen readers read out the content, and the order in which a user navigating with a keyboard can tab around the document. It's possible to create a very confusing experience by disconnecting the visual display from the source order, so we should take great care not to do so.

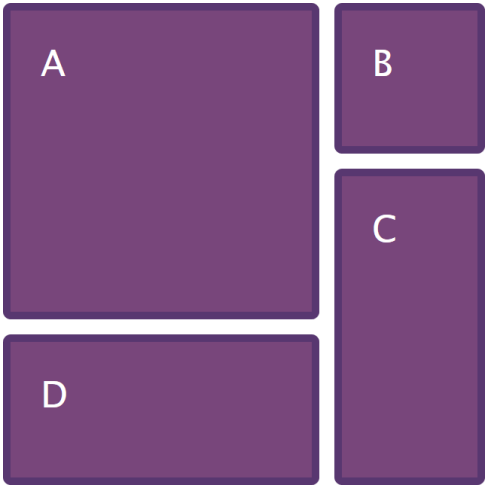


FIG 1.13: Our grid after we've used line-based placement to position the items.

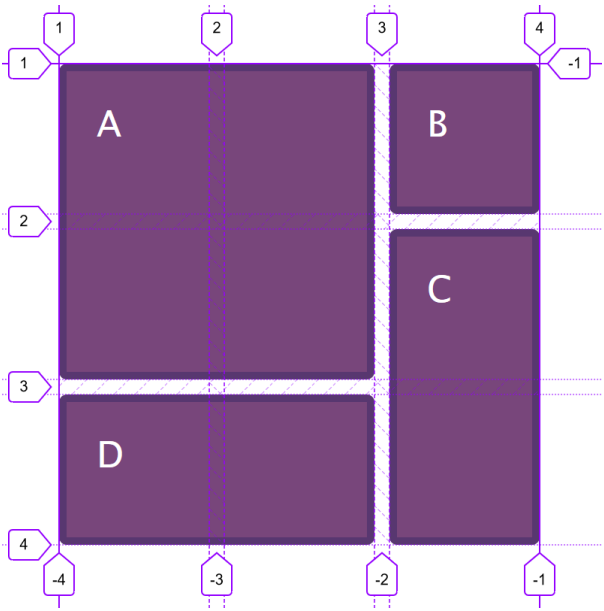


FIG 1.14: Lines highlighted using the Firefox Grid Inspector.

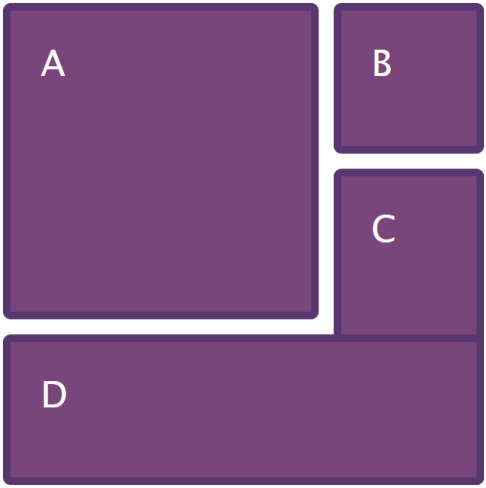


FIG 1.15: Overlapping items.

Overlapping items on the grid

When placing items using lines, we can place an item into the same cell as another item. My next example shows some elements that overlap. Items that are lower in the source display on top of items that come before them; however, we can use the `z-index` property (just as we would with absolute positioning) to change the stacking order of items (**FIG 1.15**).

Line-based placement and the `span` keyword

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

The layout shown in **FIG 1.13** could also be written like this:

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

```
}
.b {
    grid-row: 1;
    grid-column: 3;
}
.c {
    grid-row: 2 / span 2;
    grid-column: 3;
}
.d {
    grid-row: 3;
    grid-column: 1 / span 2;
}
```

Code example: http://bkaprt.com/cgl-2/01-13

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 3," I say, "Start at column line 1 and span 2 column tracks." The result is the same.

I've 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:
       [main-start col1-start] 100px
```

```
    [col1-end col2-start] 100px
       [col2-end col3-start] 100px [col3-end main-
    end];}
```

Remember: *we are naming grid lines, not tracks*. In the value for `grid-template-columns` above, I've named our first line `main-start` and `col1-start`. After that comes the `100px` track size. I move across the columns, giving the lines names. Note that lines can have multiple names, which should be separated by a space.

We can then position our items using those names, instead of numbers.

```
.a {
    grid-row: 1 / 3;
    grid-column: main-start / col2-end;
}
.b {
    grid-row: 1 / 2;
    grid-column: col3-start / col3-end;
}
.c {
    grid-row: 2 / 4;
    grid-column: col3-start / col3-end;
}
.d {
    grid-row: 3 / 4;
    grid-column: main-start / main-end;
}
```

Code example: http://bkaprt.com/cgl-2/01-14

The nice thing about naming lines this way is that once we've defined a grid, we can quickly arrange things without needing to think about their numerical position. For example, we can define `sidebar-start` and `sidebar-end` and feel confident that any element positioned there will sit in the sidebar area.

GRID TEMPLATE AREAS

The final method for creating and positioning items on the grid involves using *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.

My HTML consists of 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 these areas 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; }
```

Now I just need to declare my grid on the wrapper like I did before. This time, though, I also use `grid-template-areas` to define the layout using a kind of ASCII-art syntax.

```
.wrapper {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-areas:
  "header  header header"
  "sidebar content content"
  "footer  footer  footer";
}
```

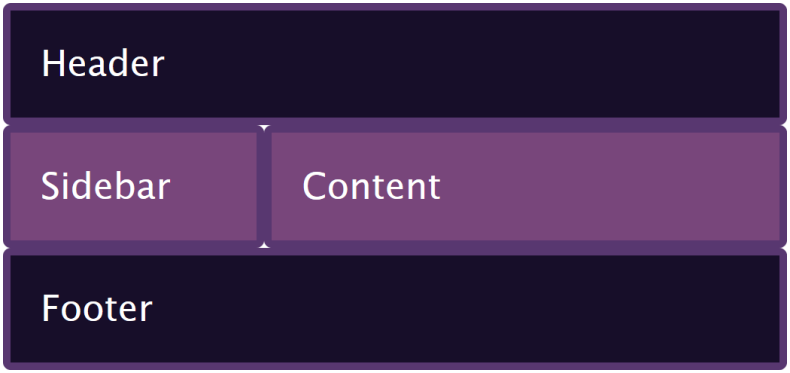


FIG 1.16: A simple layout using `grid-template-areas`.

Code example: http://bkaprt.com/cgl-2/01-15

Repeating the name of an area causes the content to span those cells. That's all we need to do to lay out a page using the CSS Grid Layout Module.

There are a few ground rules to remember when using this method. For starters, every cell on the grid must be filled. If we want to leave a cell empty, we need to use a period (`.`) For example, if we only want the footer to sit under the content, leaving an empty cell below the sidebar, our code should look like this:

```
.wrapper {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-areas:
  "header  header header"
  "sidebar content content"
  ".  footer  footer";
}
```

If we want to line up the names of the cells more neatly, we can use multiple periods. We can use additional spaces between the names for the same purpose.

```
.wrapper {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-areas:
  "header  header header"
  "sidebar content content"
  "…….  footer  footer";
}
```

Also, the areas we create must be a complete rectangle—no Tetris-style pieces. And the rectangle must be connected; we can't use this as a way to duplicate content. If we make an incorrect grid, then the whole value will be invalid, and we won't get a layout. If you find your layout isn't working, make sure every cell is filled.

No clearing is required. We can add as much or as little content as we want 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 need for any weird hacks!

Finally, we don't have to add any additional class names to our document to describe the number of columns or rows an element spans. We can simply position the elements using the classes already applied to describe the content.

All of this explains why I love the CSS Grid Layout Module so much. Read on to see more examples and discover what else is possible with 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 a few 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 (**FIG 2.1**).

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

```
<div class="wrapper">
<header class="mainheader">
  <h1>Excerpts from the book <cite>The Bristol
Royal Mail</cite></h1>
</header>
<article class="content">
  <h1>Post letter boxes: position, violation,
peculiar uses</h1>
  <div class="primary">
    <p>[code omitted for brevity]</p>
  </div>
  <aside>
    <p>[code omitted for brevity]</p>
  </aside>
</article>

<aside class="sidebar">

</aside>

<footer class="mainfooter">
  <p>[code omitted for brevity]</p>
</footer>
</div>
```

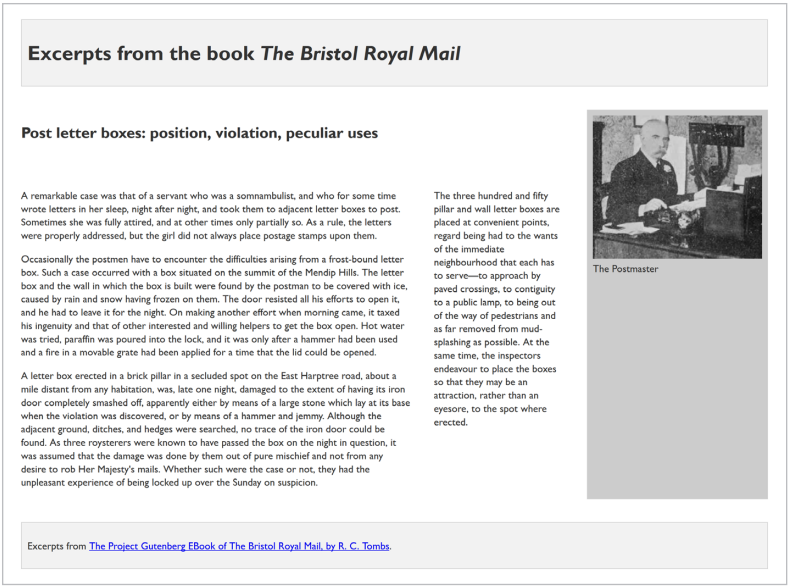


FIG 2.1: A simple three-column layout.

If we take a look at this in the browser, we'll see the content displayed in the 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 two columns: a `3fr`-unit column and a `1fr`-unit column. I have not added a value for `grid-template-rows`, since I simply want as many auto-sized rows as we need to accommodate our content.

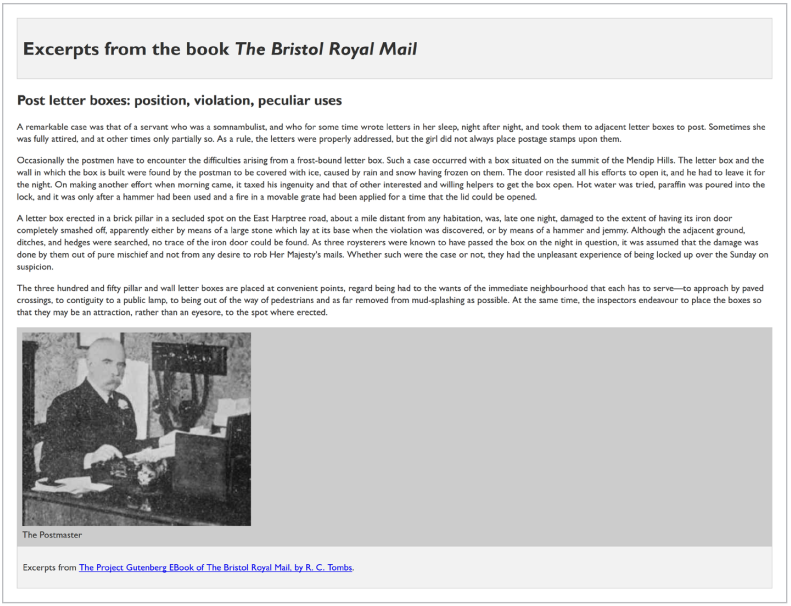


FIG 2.2: The layout before any positioning is added.

Finally, I define the layout as the value of the `grid-template-areas` property. I repeat the `header` across both 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: 3fr 1fr;
  gap: 40px;
  grid-template-areas:
    “header header”
    “content sidebar”
    “footer footer”;
```

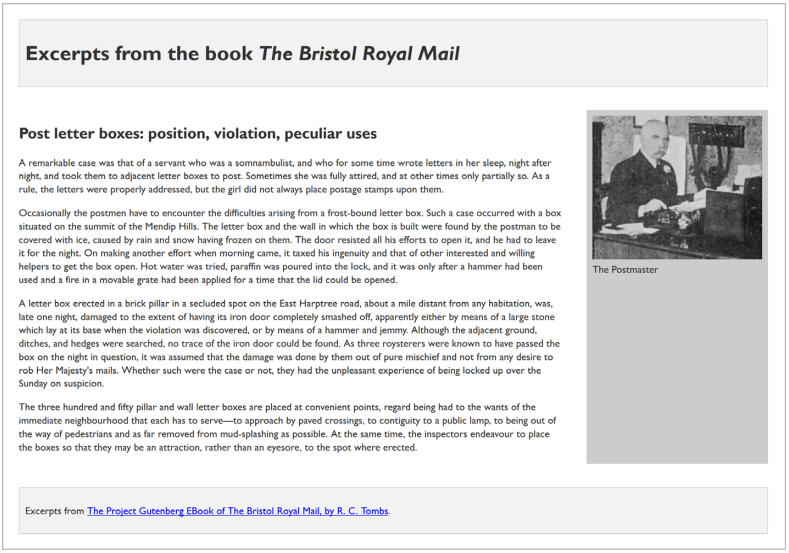


FIG 2.3: Our layout after positioning the main content areas.

```
}
```

We now have a layout taking shape, all with just a few lines of CSS (**FIG 2.3**). 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 also going to use Grid to position these, since it lets us create nested grids. I'll set up the grid areas of the nested items the same way I did with 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: 3fr 1fr;
  gap: 40px;
  grid-template-areas:
    "chapterhead ."
    "article-primary article-secondary";
  }
```

Code example: http://bkaprt.com/cgl-2/02-01

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 we want to use flexible-length units: it's tricky to get the elements in the nested grid to line up with those in the outer grid.

At the end of this book, I'll explain something that will solve this problem: the subgrid feature, which is part of Grid Level 2. For now, though, let's continue with features that have good browser support and that we can safely use in our work today.

A BOXY LAYOUT

Placement using template areas is straightforward and makes it very easy to position items into known page containers. If we want to achieve more complex layouts, however—for example, if we want to work with multiple-column grid systems—then line-based placement is the tool to use. My next example is an

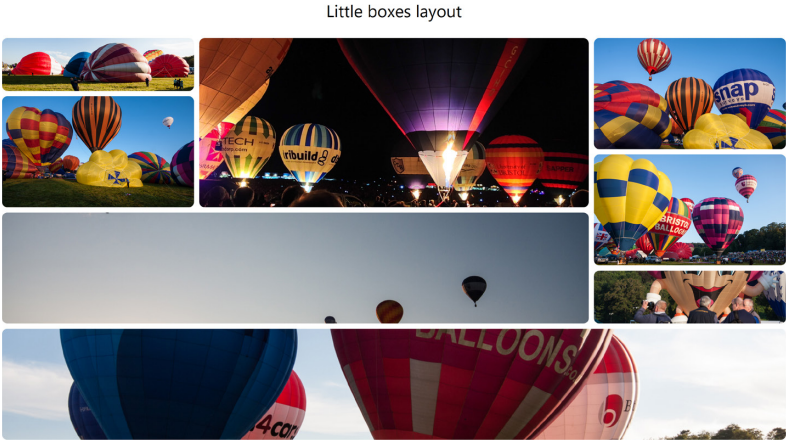


FIG 2.4: The completed boxy layout.

image layout, which could just as easily be a set of containers holding any type of content (**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="Hot-air
  balloons being inflated" class="box1">
  <img src="images/balloon2.jpg" alt="Hot-air
  balloons lit up at night" class="box2">
  <img src="images/balloon3.jpg" alt="Hot-air
  balloons launching" class="box3">
  <img src="images/balloon4.jpg" alt="Mass launch of
  hot-air balloons" class="box4">
  <img src="images/balloon5.jpg" alt="A bunch of
  hot-air balloons in the sky" class="box5">
```

```
    <img src="images/balloon6.jpg" alt="Inflated hot-
    air balloons in a field" class="box6">
    <img src="images/balloon7.jpg" alt="A hot-air
    balloon with a cartoon face on it" class="box7">
    <img src="images/balloon8.jpg" alt="Hot-air
    balloons waiting to leave the field" class="box8">
  </div>
```

I declare a grid of twelve `1fr` columns on the `.wrapper` element. This means I'm creating a grid containing thirteen column lines in total. For the rows, I start with a row whose height value is `auto` for my heading. I then create fixed-size tracks, each `100px`. Since I'm only displaying images here, the fixed height isn't a problem. My images all use the `object-fit` property with a value of `cover`, so any excess image will be cropped.

```
.wrapper {
  display: grid;
  grid-template-columns: repeat(12, 1fr) ;
  grid-template-rows: auto;
  grid-auto-rows: 100px;
  gap: 10px;
}
```

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: 1 / -1;
  grid-row: 1;
}
```

I'm using `-1` as the value for `grid-row-end`. This means that the header will end at the end column line of the explicit grid.

If we look at this layout in a browser, we can see that the images have been automatically slotted into available cells on

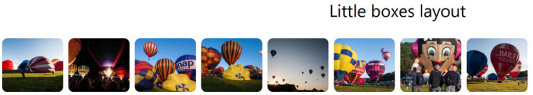


FIG 2.5: Images automatically placed one by one into each consecutive cell of the grid, before we start to lay them out.

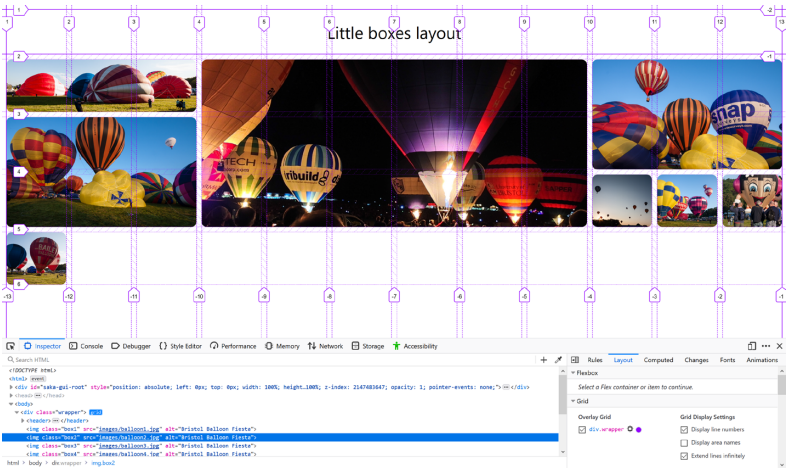


FIG 2.6: Our layout in progress, using the Grid Inspector to position items.

the grid, making them tiny (**FIG 2.5**). We can now start to place those images using line-based placement.

Using Firefox Developer Tools to highlight the lines of the grid makes creating our layout straightforward; it shows us a visual representation, with line numbers. We can even use the Grid Inspector to try out different placements of images before copying the values into our CSS (**FIG 2.6**).

Notice, as we add a rule at a time, how the remaining images continue to automatically place themselves, never overlapping. Grid is trying to give us a reasonable layout of the items as we work.

My completed set of rules looks like this:

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

    .box2 {
        grid-column: 4 / 10;
        grid-row: 2 / 5;
    }

    .box3 {
        grid-row: 2 / 4;
        grid-column: 10 / -1;
    }

    .box4 {
        grid-row: 3 / 5;
        grid-column: 1 / 4;
    }

    .box5 {
        grid-row: 5 / 7;
        grid-column: 1 / 10;
    }

    .box6 {
        grid-row: 4 / 6;
        grid-column: 10 / -1;
    }

    .box7 {
        grid-row: 6 ;
        grid-column: 10 / -1;

    }

    .box8 {
        grid-column: 1 / -1;
        grid-row: 7 / 9;
    }
```

Code example: http://bkaprt.com/cgl-2/02-02

We now have the boxy grid layout. Since this is a collection of images, we don't need to worry so much about source order. If this were a set of items that a user needed to tab between, though, we'd want to take care not to disconnect the visual view from the logical source order of our content when positioning elements.

LAYERING ITEMS ON THE GRID

If you're as old as I am, you may remember the days of using tables for layout. Conceptually, using Grid is a lot like using a table, but with a key difference: the table is not defined semantically, but in CSS. This means 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, as we learned in Chapter 1. I'm going to use this to create a detail view for an image.

First, I'll use the `box-shadow` property on `img.overlay` to make sure the selected image has a shadow. Adding a class of `overlay` to an image adds a shadow to it.

```
img.overlay {
  box-shadow: 10px 10px 20px 0px rgba(0,0,0,0.75);
}
```

Next, I'll redefine the grid for an image with a class of `overlay` to place the image on top of the other images. Because this `overlay` class comes last in the CSS, it will override any previous positioning. However, I also need to add a `z-index` value to make this element display on the top of the stack. Otherwise, images arriving later in the source will display on top.

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

```
    grid-column: 2 / -2;
    grid-row: 3 / 8;
}
```

The image now displays on top of the other images (**FIG 2.7**).

Code example: http://bkaprt.com/cgl-2/02-03

As a finishing touch, I've added a class of `.active` to the grid container so I can easily lower the opacity of images that don't have a class of `.overlay` (**FIG 2.8**).

```
.active img:not(.overlay) {
  opacity: .4;
}
```

This is essentially a very simple grid—and yet we're able to create some interesting layout patterns with it using very few lines of code. Experimenting with what is now easily possible is one of the most fun things about starting to learn Grid Layout, and I encourage you to play around and create interesting designs of your own.

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 of the specification. Next, we'll see how the Module makes it easier for us to control our layouts at multiple breakpoints.

FIG 2.7: Now that we've defined a `box-shadow` and a `z-index` value on `img.overlay`, the active image sits on top of the other images.

FIG 2.8: Fading out the inactive images.

3

CSS GRID LAYOUT AND RESPONSIVE DESIGN

IF YOU'VE MADE IT THIS FAR, you probably already have an inkling of how useful CSS Grid Layout is when working with responsive layouts. With float-based layouts, we were constrained by the need to give everything a width, and to painstakingly make sure our boxes didn't add up to more than a hundred percent across a row. With a responsive design, this often meant adding many breakpoints using media queries so we could adjust our layout.

GRID AND MEDIA QUERIES

Grid and Flexbox have flexibility built into them, which often translates into needing fewer media queries and breakpoints. That doesn't mean Grid doesn't play well with media queries, though. In this chapter, we'll put their compatibility to the test by creating responsive versions of the layouts we've worked on so far.

Let's start with the three-column layout using `grid-template-areas` from Chapter 2 (**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;
    grid-template-areas:
      "header"
      "sidebar"
      "content"
      "footer";
  }

  .content {
    display: grid;
    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` (**FIG 3.2**). At this breakpoint, I'm going to leave the inner grid alone; three columns would feel a little cramped.

```
@media only screen and (min-width: 700px) {
  .wrapper {
    grid-template-columns: 3fr 1fr;
    gap: 40px;
    grid-template-areas:
      "header header"
      "content sidebar"
      "footer footer";
```

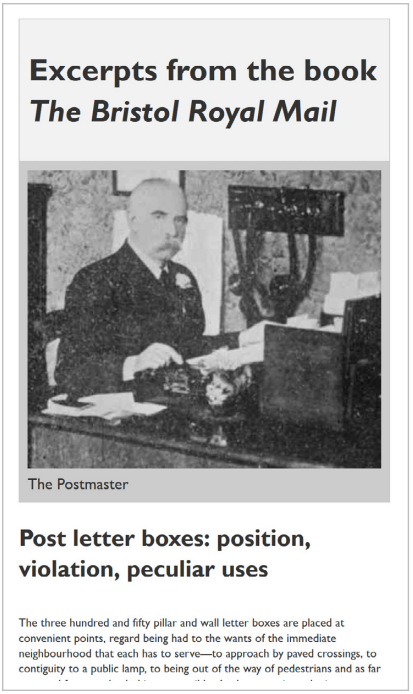


FIG 3.1: Our layout at 460 pixels.

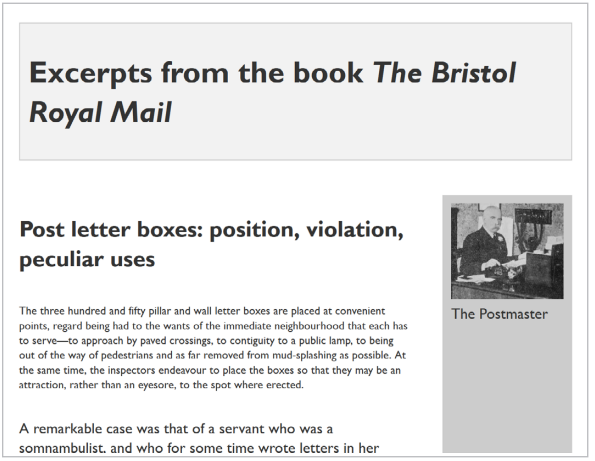


FIG 3.2: Our layout goes to two columns as we reach the 700-pixel breakpoint.

```
  }
}
```

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 {
    grid-template-columns: 3fr 1fr;
    grid-template-areas:
      "chapterhead ."
      "article-primary article-secondary";
  }
  article aside { font-size: 100%;}
}
```

Code example: http://bkaprt.com/cgl-2/03-01

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.

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.

The layout is built on a twelve-column grid. I'm going to keep that grid, which means that at narrow widths, the column tracks will be very narrow, and items will need to span more tracks (FIG 3.3).

Outside of any media queries, I add this CSS:

```
.wrapper > img {
  grid-column: 2 / -2;
}
```

FIG 3.3: Narrow-width display on a twelve-column grid.

All this does is make every direct image child of the grid container span from the second line to the second-to-last line. Note that you can count backward from the end line of the explicit grid, which is -1.

At 600 pixels, I continue taking advantage of auto-placement, and set the column and row start to `auto`—which places it wherever it naturally goes when automatically placed. However, I span the end column line six tracks, and the row line two.

FIG 3.4:
Mid-width display.


```
@media only screen and (min-width: 600px) {
  .wrapper > img {
    grid-column: auto / span 6;
    grid-row: auto / span 2;
  }
}
```

After that, we're back to the layout we created in the last chapter, wrapped in media queries with a 1000-pixel minimum. A tiny amount of code makes this layout responsive.

Of course, we can also redefine our grid for different breakpoints, or use any combination of these methods in one layout. Grid Layout and Flexbox give us the flexible-grids part of responsive design; media queries give us the ability to change the layout at different breakpoints. Grid makes it easier to do that second part, too.

4

GRID, ANOTHER TOOL IN OUR KIT

THE CSS GRID LAYOUT MODULE works alongside other layout methods to bring CSS layout up to date. Together with the Flexible Box Layout Module (Flexbox) and media queries, Grid can help us achieve the kind 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 over the other (in an ideal world where both have support)? On the www-style mailing list, Tab Atkins offered 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.*

We can better grasp this concept of one versus two dimensions through a very simple demo. In the first example below, I use Flexbox to lay out a set of five items. I've set `flex-wrap` to `wrap`, and the items display in two flex lines. The first line contains three items; the second line contains two. On the second line, the items grow to fill the space (**FIG 4.1**).

```
.wrapper {
  display: flex;
  flex-wrap: wrap;
```

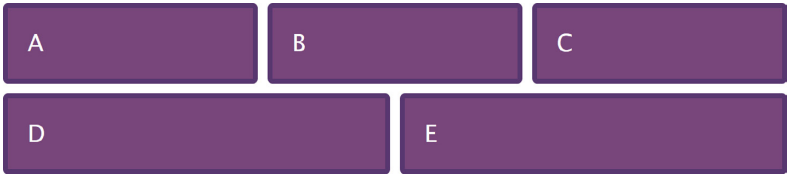


FIG 4.1: Boxes laid out with Flexbox—a one-dimensional layout.

```
}

.box {
   flex: 1 1 200px;
}
```

Code example: http://bkaprt.com/cgl-2/04-02

In Flexbox, we control one line at a time; space distribution happens per line. In other words, items don't line up in columns and rows. This is one-dimensional layout.

If we look at a similar example in Grid (again with five items), our second line contains two items, but they're lined up in columns as well as in rows. This is two-dimensional layout (**FIG 4.2**).

```
.wrapper {
   display: grid;
   grid-template-columns: repeat(auto-fill,
  minmax(200px, 1fr));
}
```

Code example: http://bkaprt.com/cgl-2/04-03

We can better understand the choices in play by creating a layout that combines Grid with other methods. Let's make 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` (**FIG 4.3**).

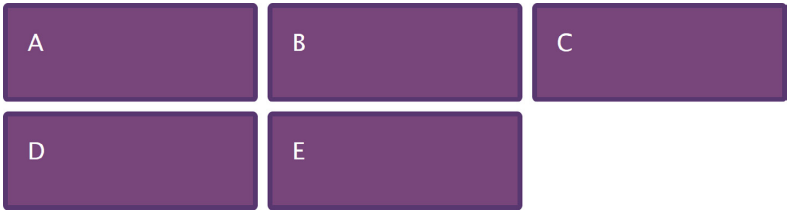


FIG 4.2: Boxes laid out with Grid—a two-dimensional layout.

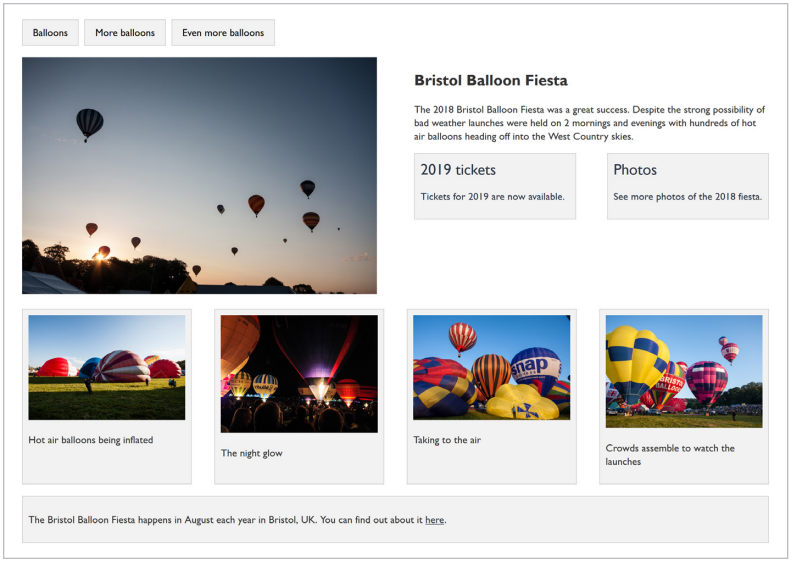


FIG 4.3: A layout combining Grid with other methods.

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

```
<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>[code omitted for brevity]</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>
```

Here, 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 a linearized view (**FIG 4.4**).

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.

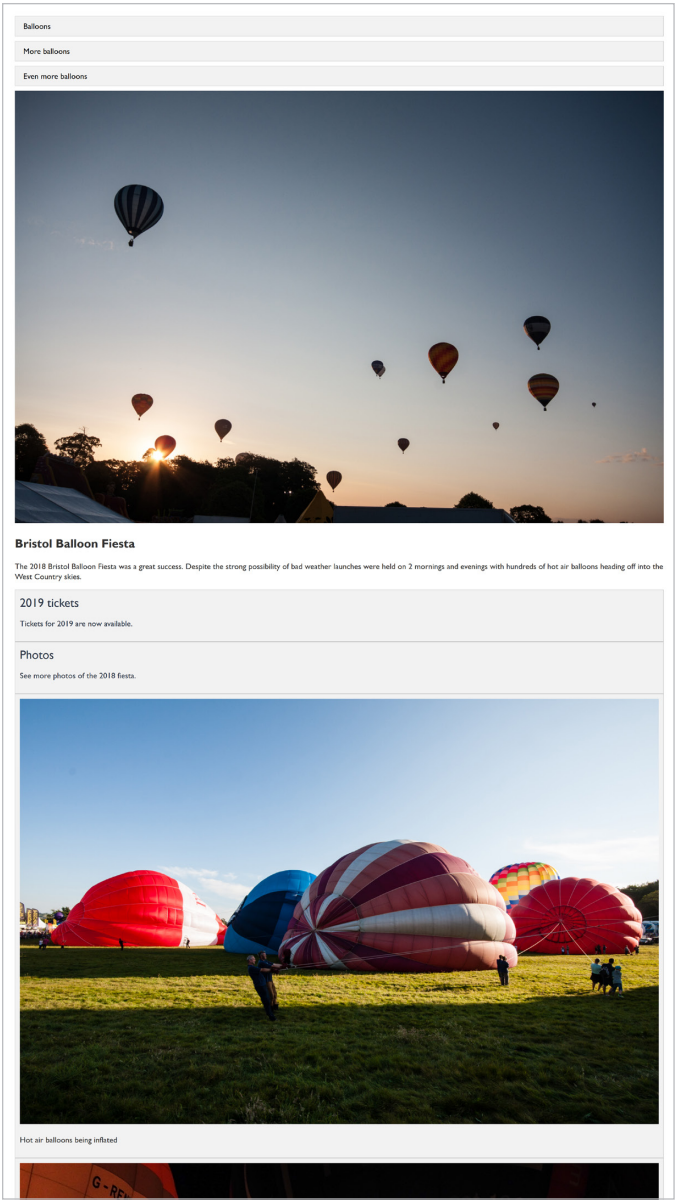


FIG 4.4: Our layout prior to adding CSS to position elements.

```
@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; }

  .wrapper {
    display: grid;
    width: 90%;
    margin: 0 auto 0 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` (**FIG 4.5**).

```
@media only screen and (min-width: 760px) {
  .wrapper {
    grid-template-columns: 1fr 1fr;
    column-gap: 5%;
    grid-template-areas:
      “header header”
      “featurepull feature”
      “secondary secondary”
      “footer footer”;
}
```

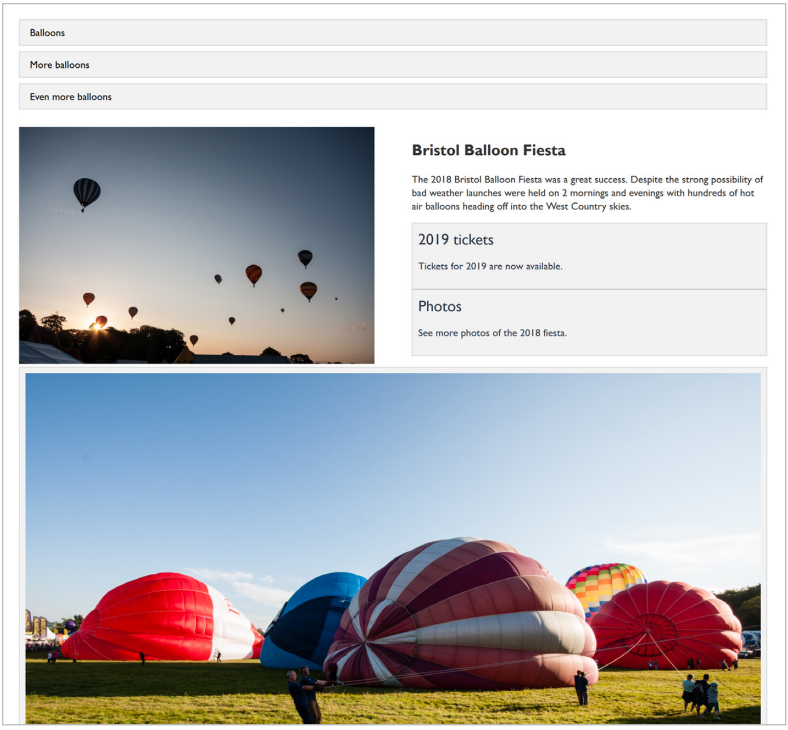


FIG 4.5: Our layout after positioning the main page areas.

At this point, we have a number of elements on the page that haven't yet been laid out to match the initial design (**FIG 4.3**). 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.

We could achieve the desired result with Grid, but Flexbox is probably a better fit for such small interface elements. Flexbox not only has the ability to wrap rows, but also has other useful features for 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.

Our simple layout is now complete and resembles the first design (**FIG 4.3**).

Code examples: http://bkaprt.com/cgl-2/04-04 and http://bkaprt.com/cgl-2/04-05

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 merely for CSS purposes. Here's the deal: we can often use *either* Flexbox or Grid for any particular component. Don't worry about it too much. If you find yourself struggling to get the layout you want, try the other method!

5

WHAT'S NEXT FOR GRID?

MY GREATEST HOPE FOR THIS BOOK is that it will give you a good introduction to CSS Grid Layout and encourage you to explore further. In this final chapter, I'd like to take a look at a feature currently being designed and implemented for the next level of the specification.

GRID LEVEL 2 AND SUBGRID

CSS specifications are developed in *levels*. Think of a level as a bit like a feature release of a product. A new level contains everything in the previous level, plus some extra features or enhancements to features. Level 2 of the Grid specification was quickly embarked on to specify a feature that had been dropped from Level 1 due to complexity. That feature is *subgrid*, and I touched on why it would be useful in Chapter 2. Let's take a closer look.

Only the direct children of a grid container become grid items. The children of the grid items return to displaying in normal flow. In the following example, the grid container has three direct children: a `div`, a `ul`, and a third `div`. Let's start with a three-column layout and place those items on the grid (**FIG 5.1**).

```
.wrapper {
  display: grid;
  grid-template-columns: 2.5fr 1fr .5fr;
  gap: 20px;
}

.box1 {
  grid-column: 1;
  grid-row: 1;
}

.box2 {
  grid-column: 2 / 4;
  grid-row: 1;
}
```

```
.box3 {
  grid-column: 1 / -1;
  grid-row: 2;
}
```

Code example: http://bkaprt.com/cgl-2/05-01

If we want the list items to be displayed using Grid Layout, we need to create a completely new grid, with its own columns and rows. In the case of a flexible, content-based grid, this may mean that the items in the list do not line up neatly with the outer grid.

With subgrid, we can make the unordered list a grid, but specify the value of `grid-template-columns` to be `subgrid`. This opts the tracks into the parent grid. They will now line up with any items positioned as direct children of the parent grid (**FIG 5.2**).

```
.box3 {
  grid-column: 1 / -1;
  grid-row: 2;
  display: grid;
  grid-template-columns: subgrid;
}
```

Code example: http://bkaprt.com/cgl-2/05-02

This makes it possible to use correct semantic markup, even with multiple levels of nesting, and lets our layout use a grid defined on an overall parent element.

At the time of writing, one browser (Firefox) actively implements subgrid. I hope that in the very near future, subgrid will be usable in more browsers.

Offering feedback on the specification

When I wrote the first edition of *Get Ready for CSS Grid Layout*, one of my aims was to introduce the emerging specification so that more designers and developers would offer feedback on it.

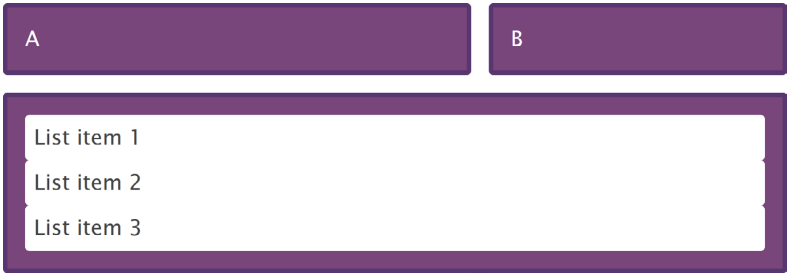


FIG 5.1: Indirect children are not part of the grid layout.

FIG 5.2: The `subgrid` value allows indirect children to participate in the grid layout.

And indeed, a comparison of the first and second versions of the spec reveals a feature that directly benefited from the web community's feedback.

The `gap` properties were added to the spec right before the first edition of this book came out. I had been pushing for the inclusion of a proper way to do gutters almost for as long as I had been using Grid Layout. Whenever I talked and wrote about Grid, I kept introducing this issue and getting enthusiastic responses from other developers who felt as strongly about it as I did. The feature was added to the spec in time for me to put a note into the first edition about it, but all of my examples still used small tracks as gutters. The result was that auto-placement was not useful—Grid would try to place an item into those tiny gaps. The examples in this edition have been updated to take advantage of the `gap` properties, and to show layouts that were impossible back when we needed to use tracks for our gutters.

Convinced of the power of community feedback, I'd like to conclude this second edition with the same call to action. I want to urge you to provide feedback for web-platform features that are still in the early stages of development. Although CSS Grid Level 1 has shipped in browsers, and Level 2 is currently being implemented, new features for CSS are continually being designed. You can—and should!—comment on them.

The CSS Working Group discussion has now moved to GitHub. You can view issues, comment on them, or raise your own questions about any specification. Many of the discussions are heavily technical, but sometimes feedback is requested on terminology, or on whether or not developers actually use a certain feature. This is crucial: developers sometimes complain that a specification does not meet their needs, so it's imperative that they get involved at a point when their concerns 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 start offering feedback on the specification or logging bugs against browser implementations. We're all busy, but contributing to the web platform can truly be seen as helping our collective future selves. Our contributions have the potential to make the platform better not only for us, but for everyone who uses the web.

RESOURCES

Read the specification to find out more about CSS Grid Layout and to stay current with browser support and developments.

Search for "css-grid" in the CSSWG GitHub repo issues to read and participate in current discussion on the Grid spec.

Grid by Example, my resource site on CSS Grid Layout, is a collection of step-by-step examples and experiments with Grid.

ACKNOWLEDGMENTS

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, the developers at Igalia, and Jen Simmons. 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/

00-02 https://caniuse.com/#feat=css-grid

Chapter 1

01-01 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-basic.html

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

01-03 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-gaps.html

01-04 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-auto-rows.html

01-05 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-minmax.html

01-06 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-fr.html

01-07 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-repeat.html

01-08 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-auto-fill.html

01-09 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-auto-fill-flexible.html

01-10 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-line-based.html

01-11 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-line-based-shorthand.html

01-12 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-line-based-grid-area.html

01-13 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-line-based-span.html

01-14 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-line-based-named-lines.html

01-15 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch1-grid-template-areas.html

Chapter 2

02-01 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch2-layout.html

02-02 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch2-boxy.html

02-03 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch2-boxy-overlay.html

Chapter 3

03-01 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch3-layout.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/2e/ch4-layout.html

04-03 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch4-layout-styles.css

04-04 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch4-layout.html

04-05 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch4-layout-styles.css

Chapter 5

05-01 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch5-nosubgrid.html

05-02 https://github.com/abookapart/css-grid-layout-code/blob/master/2e/ch5-subgrid.html

05-03 https://github.com/w3c/csswg-drafts/issues

Resources

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

06-02 https://github.com/w3c/csswg-drafts/issues

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 is half of the web development firm edgeofmyseat.com and editor-in-chief of *Smashing Magazine*. She is a member of the CSS Working Group and a Google Developer Expert.

Rachel has been working on the web since 1996 and writing about the web for almost as long. She's written several books including *Get Ready for CSS Grid Layout*, 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!