2 Designing a layout using CSS Grid

This chapter covers

- Exploring grid tracks and arranging our grids
- Using the `minmax` and `repeat` functions in CSS Grid
- Working with the fraction unit, which is unique to CSS Grid
- Creating template areas and placing items in the areas
- Considering accessibility when using grids
- Creating gutters between columns and rows within grids

Now that we have a basic understanding of how CSS works, we can begin exploring our options for laying out HTML content. In this chapter, we'll focus on layout with grids.

.1 CSS Grid

A *grid*, in this sense, is a network of lines that cross to form a series of squares or rectangles. Now supported by all major browsers, CSS Grid has become a popular layout technique.

Essentially, a grid is made up of columns and rows. We'll create our grid and then assign positions to our items much as we place boats on a grid when playing the board game Battleship.

Although grid layouts are sometimes compared with tables, they have different uses and fulfill different needs. Grids are for layouts, whereas tables are for tabular data. If the content being styled is appropriate for a Microsoft Excel sheet, it's tabular data and should be placed in a table.

In the mid-2000s, we used tables for layouts, and sometimes we still need to. (Emails, for example, sometimes require the use of tables for layout, as they support only a subset of CSS styles.) On the web, however, this technique

is considered to be bad practice because it leads to poor accessibility and lack of semantics. Now we can use a grid instead.

CSS Grid empowers us to be creative, to produce a range of layouts, and to adapt those layouts for different conditions in conjunction with media queries. We'll use a grid to style our project, and by the end of the chapter, our layout will look like figure 2.1.

Figure 2.1 Final output

Our starting HTML, in the chapter-02 folder of the GitHub repository (**https://github.com/michaelgearon/Tiny-CSS-Projects**) or in CodePen (**https://codepen.io/michaelgearon/pen/eYRKXqv**), looks like the following listing.

Listing 2.1 Project HTML

```
<body>
  <main>                                    ①
    <header>                                ②
      <img src="./img/logo.jpg" alt="">
      <h1>Sonnets by William Shakespeare</h1>
```

```
</header>
<article>                                    ②
  <h2>
    Sonnet 1
    <br><small>by William Shakespeare</small>
  </h2>
  <p>
    <span>From fairest creatures we desire increase,</span>
    ...
  </p>
</article>
<aside>                                      ②
  <section>
    <img src="./img/image-1.jpg" alt="">
    <h3>Sonnet 2</h3>
    <p>
      When forty winters shall besiege thy brow,
      <br>And dig deep trenches in thy beauty's field, ...
    </p>
    <a href="">Read more of Sonnet 2</a>
  </section>
  <section>
    <img src="./img/image-2.jpg" alt="">
    <h3>Sonnet 3</h3>
    <p>
      Look in thy glass and tell the face thou viewest,
      <br>Now is the time that face should form another, ...
    </p>
    <a href="">Read more of Sonnet 3</a>
  </section>
</aside>
<section class="author-details">             ②
  <h3>
    <small>About the Author</small>
    <br>William Shakespeare
  </h3>
  <p>English playwright, poet, ...</p>
</section>
<section class="plays">                      ②
  <img src="./img/play.jpg" alt="">
  <h3>Checkout out his plays:</h3>
  <ul>
    <li><a href="">All's Well That Ends Well</a></li>
```

```
          ...
        </ul>
      </section>
      <footer>                                    ②
        <p>Want to read more ...</p>
      </footer>
    </main>
  </body>
```

① The container for our project

② The child items within our container

We also have some starting CSS (listing 2.2) to guide us as we start to place our HTML elements in a grid format. We won't worry about these preset styles (such as margin, padding, colors, typography, and borders) in this chapter. Those concepts are covered in other parts of the book because we want to focus on the layout for this project.

Listing 2.2 Starting CSS

```
  body {
    margin: 0;
    padding: 0;
    background: #fff9e8;
    min-height: 100vh;                            ①
    font-family: sans-serif;
    color: #151412
  }
  main { margin: 24px }
```

```
img {
  float: left;                                    ②
  margin: 12px 12px 12px 0
}
main > * {                                        ③
  border: solid 1px #bfbfbf;                      ④
  padding: 12px;
}
main > *, section { display: flow-root }          ⑤
p, ul { line-height: 1.5 }
article p span { display: block; }
article p span:last-of-type,                      ⑥
article p span:nth-last-child(2) {                ⑥
  text-indent: 16px                               ⑥
}                                                 ⑥
.plays ul { margin-left: 162px }                  ⑦
```

① The background covers the whole page even when the window is longer than the content.

② Allows text to wrap around the image

③ Asterisk and child combinator selects any and all immediate children of main.

④ border points out sections to be positioned via a grid.

⑤ Prevents images from bleeding out of their containers

⑥ Indents the last two lines of the sonnet

⑦ Indents the list; otherwise, bullets are right up against the image.

We change the font from `serif` to `sans-serif`, and we increase the margin between the boundary of the browser window and the container by using `margin`. We also float images to the left and adjust the line heights, typography, and padding.

Note that we added a border and some padding to the immediate children of the `main` element to help us define our layout. We'll remove those elements later in the project. Our starting point looks like figure 2.2.

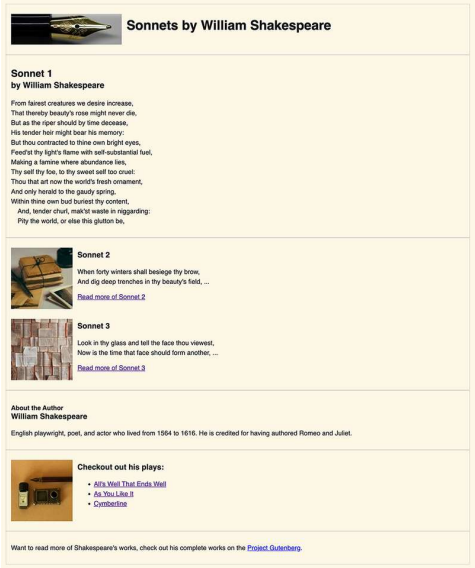

Figure 2.2 Starting point

CSS grids are a way to place items on a 2D layout: horizontal (x-axis) and vertical (y-axis). By contrast, the flexbox (covered in chapter 6) is single-axis-oriented. It operates only on the x- or y-axis, depending on its configuration.

We can use both CSS Flexbox and CSS Grid to align and lay out items on a web page. But as we go through the chapter, we'll find that one of the benefits of Grid over Flexbox is that it allows us to divide a page into regions and create complex layouts with relative ease.

First, we'll set up our grid. Then we'll look at ways to alter how our grid behaves based on window size.

.2 Display grid

The first part of arranging a grid is setting the `display` value to `grid` on the parent container item. When creating a grid layout, we can use one of two values:

- `grid` —Used when we want the browser to display the grid in a block-level box. The grid takes the full width of the container and sets itself on a new line.
- `inline-grid` —Used when we want the grid to be an inline-level box. The grid sets itself inline with the previous content, much like a `<span>`.

We'll use the value `grid` for our layout, as shown in listing 2.3.

Difference between block-level and inline-level box

In HTML, every element is a box. A block-level box says that an element's box should use the entire horizontal space of its parent element, therefore preventing any other element from being on the same horizontal line by default. By contrast, inline elements can allow other inline elements to be on the same horizontal line, depending on the remaining space.

Listing 2.3 Setting the display to `grid`

```
main {
  display: grid;
}
```

If we preview this code in the browser, we notice that nothing has changed visually because the browser by default displays the direct child items in one column. Then the browser generates enough rows for all of the child elements.

Using the developer tools in our browser (figure 2.3), we see that, programmatically, the grid has been created even though the layout has not changed visually. To view the underlying grid in most browsers, we can right-click the web page and choose Inspect from the contextual menu. In the Inspect window in Mozilla Firefox, when we select the parent container, we see two things to indicate the layout is now a grid:

- Purple lines around each direct child item.

- In the HTML, an icon called grid in the `<main>` element. When we click the grid icon next to `<main>`, the layout panel shows us our grid structure.

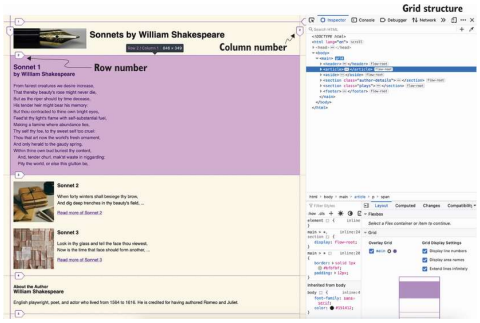

Figure 2.3 Development tools in Firefox

We can follow similar steps in Google Chrome or Apple's Safari.

.3 Grid tracks and lines

When the CSS Grid Layout Module was introduced, it brought in new terminology to describe laying out items. The first of those terms is *grid lines.* Lines run horizontally and vertically, and they're numbered starting from 1 in the top-left corner. On the opposite side to the positive numbers are the negative numbers.

Writing mode and script direction

The number assigned to each line depends on the writing mode (whether lines of text are laid out horizontally or vertically) and the script direction of the component. If the writing mode is English, for example, the first line on the left is numbered 1. If the language direction is set to right-to-left because of the language, for example, Arabic (which is written from right to left), line 1 would be the farthest line to the right.

The spaces between the grid lines are known as *grid tracks,* and they're made up of columns and rows. Columns go from left to right, and rows go from top to bottom. A *track* is the space between any two lines on a grid. In figure 2.4, the highlighted track is the first row track in our grid. A column track would be the space between two vertical lines.

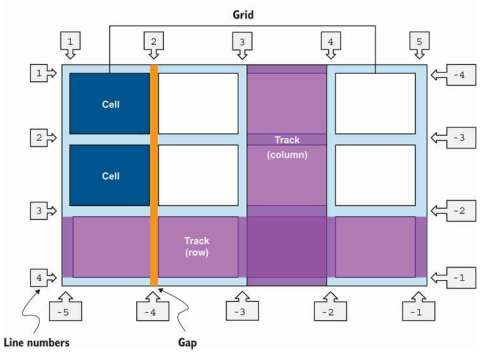

Figure 2.4 Grid structure based on English as the writing mode, with the direction set to left-to-right

Within each track are grid cells. A *cell* is the intersection of a grid row and a grid column.

We can use the `grid-template-columns` and `grid-template-rows` properties to lay out our grid. These properties specify, as a space-separated track list, the line names and track sizing functions of the grid. The `grid-template-columns` property specifies the track list for the grid's columns, and `grid-template-rows` specifies the track list for the grid's rows.

Before we set our columns, we need to understand a few concepts that are specific to CSS Grid.

2.3.1 Repeating columns

To save repetition in your code, you can use the `repeat()` function to specify how many columns or rows you need.

DEFINITION A *function* is a self-contained, reusable piece of code that has a specific role. Functions exist in other programming languages, such as JavaScript. Sometimes, we can pass one or more values to the function; these values are referred to as *parameters*. To pass values to a function, we place them in parentheses. We can't create our own functions in CSS; instead, we use the built-in functions that CSS offers.

The `repeat()` function needs two values that are comma-separated. The first value indicates how many columns or rows to create. The second value is the sizing of each column or row.

For our project, we'll specify that we want two columns, and for the sizing of each column, we'll use the `minmax()` function. Our column defini-

tion, therefore, will be `grid-template-columns: repeat(2, minmax(auto, 1fr)) 250px;`. If we were defining the height of our rows, we'd use `repeat()` with `grid-template-rows`.

This declaration produces three columns, two of equal width using the fraction unit and one of 250 pixels. Let's look at this declaration a bit further. Notice that inside the `repeat()` function, we use the `minmax()` function.

2.3.2 The minmax() function

The `minmax(min, max)` function is made up of two arguments: the minimum and maximum range of the grid track. The World Wide Web Consortium (W3C) specification states that the `minmax` function "defines a size range greater than or equal to `min` and less than or equal to `max`" (**https://www.w3.org/TR/css-grid-2**).

NOTE To make the function valid, the `min` value (the first argument) needs to be smaller than the `max` value. Otherwise, the browser ig-

nores `max`, and the function relies only on the `min` value.

For our project, we set the `min` value to `auto` and the `max` value to `2`. Let's look at what `auto` means.

2.3.3 The auto keyword

The `auto` keyword can be used for either the minimum or maximum value within the function. When the keyword `auto` is used for the maximum value, it's treated the same as the `max-content` keyword. The row's or column's dimensions will be equal to the amount of room that the content within the row or column requires.

Although we don't use it in our project, a common use case for the `auto` keyword is making layouts that include fixed headers and footers. When we assign overflow to the area for which `auto` was set, the area shrinks and grows with the window size, as shown in figure 2.5.

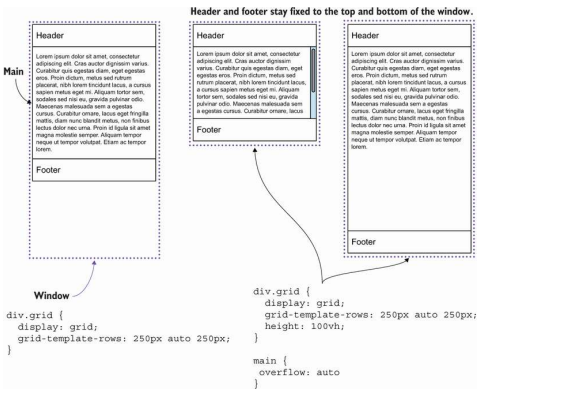

Figure 2.5 Examples of using the `auto` keyword

For our use case, in the statement `grid-template-columns: repeat(2, minmax(auto, 1fr)) 250px;`, the `auto` keyword dictates that for our first two columns, the column should be, at minimum, as wide as the element it contains. Let's take a look at the flexible length unit (`fr`) used to set our maximum width.

2.3.4 The fractions (fr) unit

The fractions (`fr`) unit was introduced in the CSS Grid Layout Module. The `fr` unit, which is unique to Grid, tells the browser how much room an HTML element should have compared with other elements by distributing the leftover space after the minimums have been applied. CSS distributes the available space equally among the `fr` units,

so the value of `1fr` is equal to the available space divided by the total number of `fr` units specified.

Let's explore what a fraction is through the tasty cake diagrams shown in figure 2.6. (Sorry if this figure makes you crave a slice of cake.)

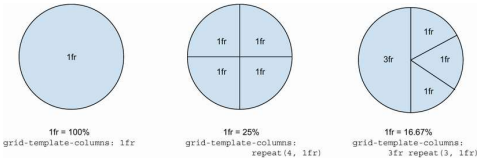

Figure 2.6 Fractions values

If you had a whole cake, it would equal 100%. From a CSS perspective, if we decided to eat all the cake, that would be 1 fraction. In our CSS, that would be the same as `grid-template-columns: 1fr`, which would be 100% of the column.

But we're friendly, so we decide to give some of our cake to four friends. We need to determine how many slices of our cake each person is going to have.

If we're fair, we can say that our cake can be divided into four equal slices. In our CSS,

this would be the same as `grid-template-columns: 1fr 1fr 1fr 1fr`. We're telling the browser to give each HTML element an equal slice of the whole thing.

But what if we decided to be sneaky and keep a larger slice for ourselves? After all, we baked the cake. We decide to take half of the cake for ourselves and divide the remaining half into three slices. To do this, we need six fractions: three fractions for our 50% of the cake and then one fraction three times to divide the other 50% of the cake.

Our CSS would be `grid-template-columns: 3fr 1fr 1fr 1fr`. So we're saying that there are six fractions total; the first column gets three of them (or 50% of the total), and then the remaining 50% should be divided equally among the other three columns. We can use the `fr` unit with the `repeat()` function to make this value easier to read, which would be `grid-template-columns: 3fr repeat(3, 1fr)`.

For our project we'll create our grid lines for the columns by adding the code in the following listing to our `main` rule.

Listing 2.4 Setting the amount of columns

```
main {
  display: grid;
  grid-template-columns: repeat(2, minmax(auto, 1fr)) 250px;
}
```

When previewed in the browser (figure 2.7), we see that now our grid has numbers set across each line. What we can do with this information is explicitly choose where to place our HTML elements within the grid based on the grid line number.

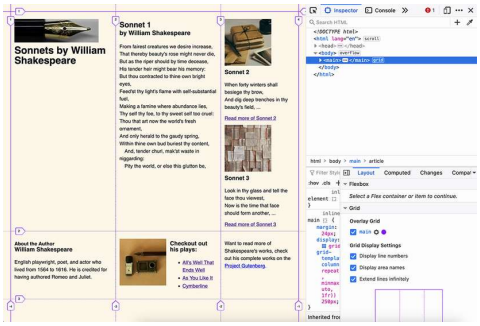

Figure 2.7 Firefox browser preview showing the grid lines and associated numbers for each line

We also notice that the browser assumed that we want to place our HTML ele-

ments within each grid cell. Rather than stacking the elements vertically, the browser filled each column cell until it ran out, created a new row, and filled in those columns. The automatic creation of extra grid cells is also known as *an implicit grid*.

Explicit versus implicit grid

When we use `grid-template-columns` or `grid-template-rows`, we're creating an explicit grid. We're clearly stating to the browser the exact amount of columns and rows this grid should have.

The implicit parts (for both rows and columns) are those that the browser creates automatically, which can happen when there are more child items than grid cells. In this case, the browser implicitly adds cells to our grid to make sure that all elements have a grid cell.

We can control implicit behavior through `grid-auto-flow`, `grid-auto-columns`, and `grid-auto-rows`.

At this juncture, we've created a grid containing three col-

umns. Two of those columns use `minmax()`, and our third column has a fixed value of `250px`. These settings give us a three-column layout. We want to distribute the main content in the first two columns and use the third one for less important content, which is why we give it less visual real estate. (On most screens, the third column will be narrower than the first two columns.)

.4 Grid template areas

If we want to set an element explicitly on a particular row and column of our grid, we have two options. First, we can use the line numbers and dictate the position of the child as follows: `grid-column: 1 / 4`. In this syntax, the first number represents where the element starts, and the second represents where it ends (figure 2.8). This example places the element in the first column, spanning the second and third. If only one number is provided, the element spans only one row or column.

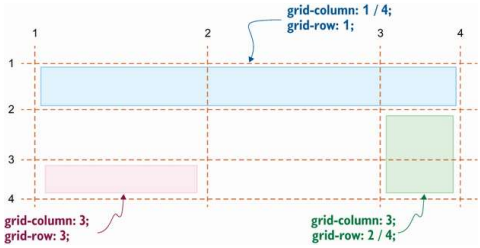

Figure 2.8 Example `grid-column` and `grid-row` syntax

To define the row, we would use the same syntax as for columns with the `grid-row` property. To place an element so that it starts on the third row and spans two rows, we would write `grid-row: 3 / 5`. The `grid-column` and `grid-row` properties are shorthand for `grid-column-start`, `grid-column-end`, `grid-row-start`, and `grid-row-end`.

Rather than deal with numbers, we can use named areas to be referenced when we explicitly place elements on the grid. To do this, we use the `grid-template-areas` property, which allows us to define how we want the web page to be laid out.

The `grid-template-areas` property takes multiple strings, each composed of the names of the areas they describe. Each string represents

a row in the layout, as shown in figure 2.8. Each name represents a column within the row. If two adjacent cells have the same name (horizontally or vertically), the two cells are treated as one area. A grid area can be a single cell, such as the area defined as `plays` in figure 2.9, but if it's more than one cell, the cells must create a rectangular shape with all cells of the same name being adjacent. You wouldn't be able to make an L shape, for example.

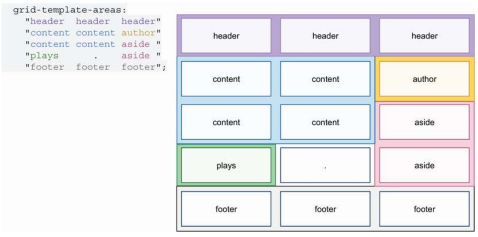

Figure 2.9 Syntax of the `grid-template-areas` property

The benefit of named areas is in the visualization of the final outcome. We'll define our `grid-template-areas` as shown in listing 2.5. Notice the dot (`.`) in the fourth row in figure 2.9. The dot is used to define a cell that we intend to keep empty. Because that cell doesn't have a name, content can't be assigned to it.

Listing 2.5 Creating our template areas

```
main {
  display: grid;
  grid-template-columns: repeat(2, minmax(auto, 1fr)) 250px;
  grid-template-areas:
  "header  header  header"
  "content content author"
  "content content aside "
  "plays       .    aside "
  "footer  footer  footer";
}
```

Although we've defined areas, the content still implicitly positions itself in each available cell, ignoring the areas we defined (figure 2.10). We need to assign our content to each of these areas.

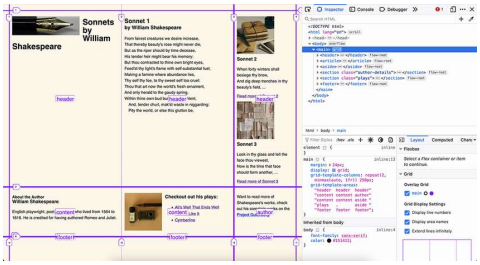

Figure 2.10 Defined grid areas shown in Firefox

2.4.1 The grid-area property

To place an element in a defined area, we use the `grid-area` property. Its value is the name we assigned in the `grid-template-areas` property. If we want the `<header>`

element to be placed inside the area we defined as `header`, for example, we would define `header { grid-area: header; }`. For our project, we set our elements on our grid as shown in the following listing.

Listing 2.6 Assigning content to the grid area

```
header { grid-area: header }
article { grid-area: content }
aside { grid-area: aside }
.author-details { grid-area: author }
.plays { grid-area: plays }
footer { grid-area: footer }
```

Now that we've explicitly defined where the content should go, the content falls into place (figure 2.11).

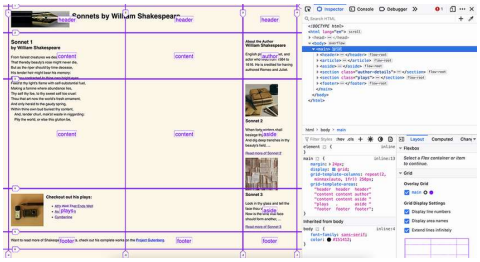

Figure 2.11 Content explicitly placed on the grid

With the layout setup, let's remove some of the styles we added for the purpose of understanding what our layout was doing. As shown in the

following listing, we remove the padding and borders of our content sections.

Listing 2.7 Removing debugging styles

```
main > * {
  border: solid 1px #bfbfbf;
  padding: 12px;
}
```

With those styles removed and the screen width narrowed (figure 2.12), the content in adjacent columns or rows appears to be closer together.

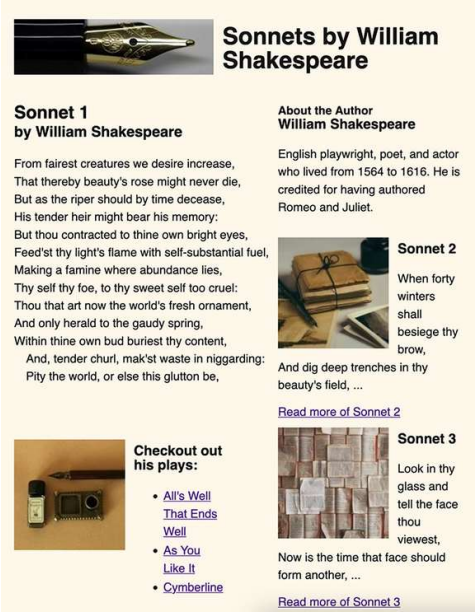

Figure 2.12 Narrow screen width

Let's add space between the areas. To accomplish this task, we will use the `gap` property.

2.4.2 The gap property

The `gap` property is shorthand for the `row-gap` and `column-gap` properties. By setting the row and column gaps, we're defining the gutters between rows and columns. *Gutters* is a term from print design, defining the gap between columns. By default, the gap between columns and rows is the keyword `normal`. This value equates to `0px` in all contexts except when it's used with the CSS Multi-Column Module, which equates it to `1em`.

When we use the `gap` property, the extra space applies only between the tracks of the grid. No gutters are applied before the first track or after the last track. To set space around the grid, we use padding and margin properties.

gap vs. grid-gap

As CSS Grid was being defined, the specification for this property was called the `grid-gap` property, but now `gap` is recommended. We may see `grid-gap` in older projects.

The `gap` property can have up to two positive values. The first value sets the `row-gap`, and the second is for the `column-gap`. If only one value is declared, it's applied to both the `row-gap` and `column-gap` properties.

For our project, we'll set a `20px` gap between our rows and columns by adding `gap: 20px` to our `main` rule. Figure 2.13 shows the gaps added to our layout. With the gaps added, let's switch our focus to adjusting our layout based on screen size.

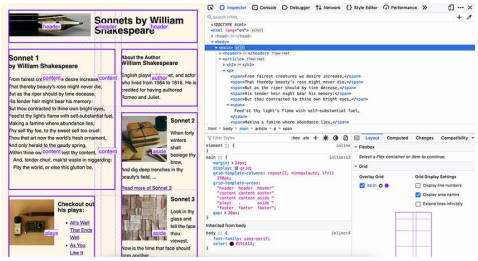

Figure 2.13 Grid layout with added gap

.5 Media queries

CSS allows us to apply styles to our layout conditionally. One type of condition is screen size. Media queries are *at-rules*: they start with the at (`@`) symbol and define the condition under which the styles they contain should be

applied. If we look at our current layout on a wide screen (figure 2.14), we notice a large amount of space in the center of the page that could be put to better use.

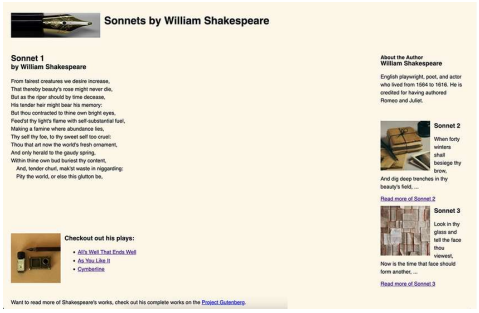

Figure 2.14 Our layout on a wide screen

Let's create a media query that targets screens wider than `955px`. The query is `@media (min-width: 955px) { }`. All the rules we place inside the curly braces (`{}`) will be applied only if the screen size is greater than or equal to `955px`.

Listing 2.8 shows our media query. We redefine our `grid-template-areas` to have a different configuration if the media-query condition is met. We also update the column sizes so that the columns have equal widths.

Listing 2.8 Creating our template areas with media queries

```
@media (min-width: 955px) {                    ①
  main {
    grid-template-columns: repeat(3, 1fr);     ②
    grid-template-areas:                       ③
    "header  header  header"                   ③
    "content author aside"                     ③
    "content plays aside "                     ③
    "footer  footer  footer";                  ③
  }
}
```

① The at-rule along with media feature

② Redefines the column sizes

③ Reconfigures where the content should be placed

Now our layout looks like figure 2.15 and figure 2.16.

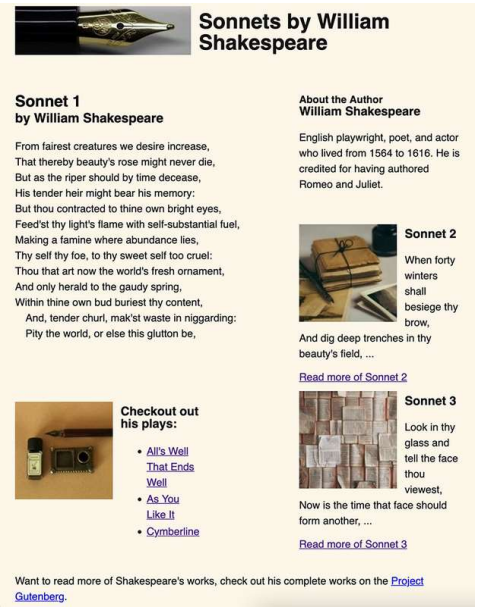

Figure 2.15 Narrow screen uses the original layout.

Using `grid-template-areas` in conjunction with media

queries allows us to reconfigure our layout with minimal code. But we must avoid some accessibility pitfalls.

Figure 2.16 Wide screen uses the layout from the media query.

.6 Accessibility considerations

When we placed our items in the grid area, we mostly kept the elements in the order in which they appeared in the HTML: the header stayed at the top, footer remained at the bottom, and the content was in a logical visual order. But what if the HTML order and the visual display order were different?

If a user is following along with a screen reader or navigating the page via the keyboard, and the programmatic order doesn't match what's being displayed, the behavior will seem to be random. This randomness will make it difficult for the user to navigate

the page and to comprehend what's going on with it. Visually changing the location of a piece of content by using a grid won't affect the order in which assistive technology presents the information to the user. The W3 Grid Layout Module Recommendations states the following about this case (**http://mng.bz/xdD7**):

Authors must use order and the grid-placement properties only for visual, not logical, re-ordering of content. Style sheets that use these features to perform logical reordering are non-conforming.

The solution is to keep the source code and the visual experience the same, or at least in a sensible order. This approach gives you both the most accessible web document and a good structure to work from. For English, this means that content and HTML should follow the same order, from top left to bottom right.

After assigning our elements to their respective areas of the grid, we should always test our page to ensure that regardless of how the user ac-

cesses the page, the order will be logical. One way to do this is to visit our page with a screen reader and tab through the elements to make sure that the tab order still works.

Some tools and extensions can help with visualizing tab order. In Firefox DevTools, for example, we can select the Accessibility tab and check the Show Tabbing Order check box, which outlines and numbers focusable elements as shown in figure 2.17. We can see that our tab order is logical and unlikely to confuse the user, so we're good to go.

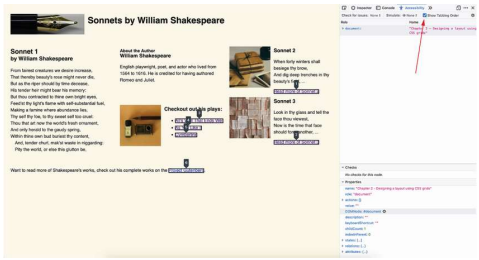

Figure 2.17 Tabbing order of HTML exposed in Firefox DevTools

Now our project is complete (figure 2.18).

Future of Grid

In this chapter, we used the CSS Grid Layout Module to create a layout that's responsive depending on the browser

width. Many aspects of the grid are still being developed and iterated, most notably subgrids, which would allow for grids within grids.

Although you can set a grid within a grid now, subgrids have the benefit of being more closely related to their parent grid. To keep an eye on future enhancements and development, check out the grid specification at **https://www.w3.org/TR/css-grid**.

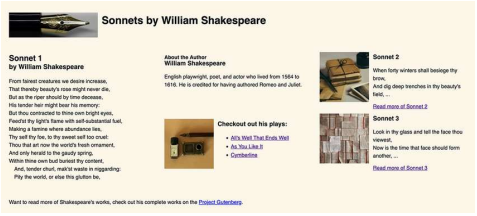

Figure 2.18 Final product on wide screen

ummary

- A grid is a network of lines that cross to form a series of squares or rectangles.
- The `display` property with a value of `grid` allows us to place items on a grid layout.

- The `display` property is applied to the parent item that contains the child elements that are to be placed on the grid.
- The `grid-template-columns` and `grid-template-rows` properties are used to explicitly define the quantity and size of the columns and rows the grid should contain.
- The flexible length (`fr`) unit is a unit of measurement that was formed as part of CSS Grid as an alternative way to set the dimension of items.
- We can use the `repeat()` function to improve code efficiency where one or more rows or columns are the same size.
- The `minmax()` function allows us to set two arguments: the minimum width a column should be and the maximum width a column should be.
- The `grid-template-areas` property allows us to define what each grid area is called. Then we can use the `grid-area` property on the child items to assign them to those named locations.

- The `gap` property adds spacing (creates gutters) between grid cells.
- The source code and the visual experience need to stay in the same logical order. When in doubt, we can use browser developer tools to check the tabbing order.