
Designing with
CSS Grid Layout

Designing with CSS Grid Layout

Cover Design: Natalia Balska

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

Table of Contents

Preface

Layout in CSS has always been a tricky task: hacking solutions using positioning, floats, and the one-dimensional flexbox has never been very satisfactory. Fortunately, there is a new tool to add to our arsenal: CSS Grid Layout>. It is an incredibly powerful layout system that allows us to design pages using a two-dimensional grid - offering the kind of fine-grained layout control that print designers take for granted!

Grid Layout's been in development for a while, but has recently been made a W3C candidate recommendation and has been added to most of the major browsers>, so is ready for prime time.

This short selection of tutorials, hand-picked from SitePoint's HTML & CSS channel, will get you up and running with Grid Layout and using it in your own sites in no time.

Conventions Used

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk.</p>
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An ↪ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-design-real
↪ -user-testing/?responsive1");
```

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Chapter 1

An Introduction to the CSS Grid Layout Module

by Ahmad Ajmi

As web applications become more and more complex, we need a more natural way to do advanced layouts easily without hacky solutions that use floats and other less burdensome techniques. An exciting new solution for creating layouts comes with the CSS Grid Layout Module.

In this introductory tutorial, I'll introduce you to this relatively new CSS feature and I'll show you using some examples how the CSS Grid Layout Module works.

What is the CSS Grid Layout Module?

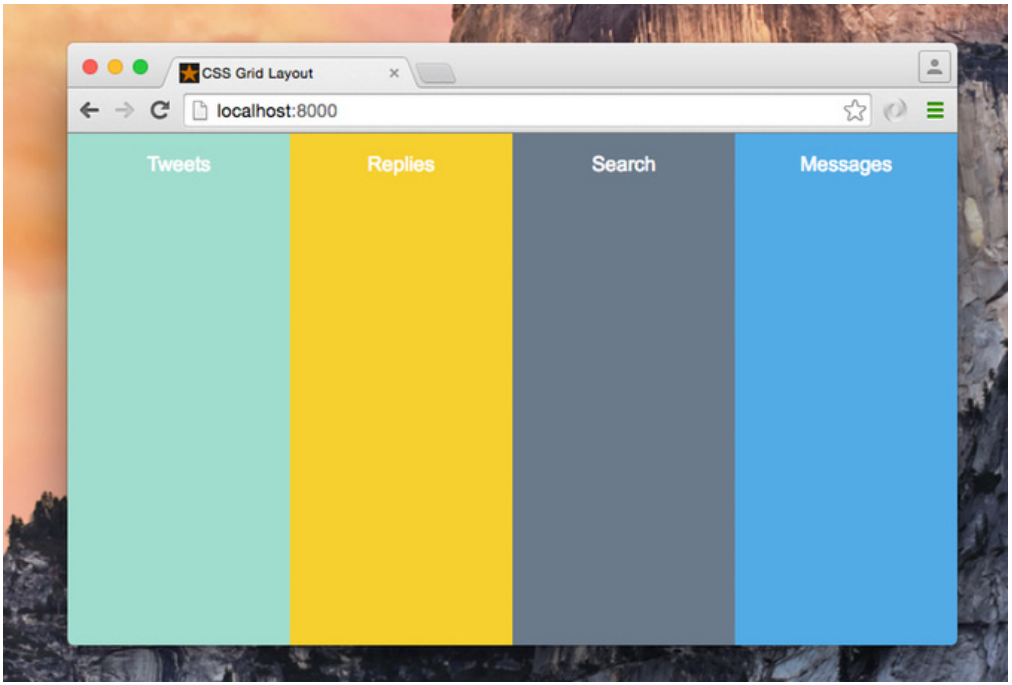
The core idea behind the Grid Layout is to divide a web page into columns and rows, along with the ability to position and size the building block elements based on the rows and columns we have created in terms of size, position, and layer.

The grid also gives us a flexible way to change the position of elements with only CSS without any change to the HTML. This can be used with media queries to alter the layout at different breakpoints.

A Grid Layout Example

Let's start with an example to see the power of Grid Layout, and then I'll explain some new concepts in more detail.

Imagine you want to create a Twitter app with four full height columns layout (Tweets, Replies, Search, and Messages), something abstracted and similar to the screenshot below.



Here is our HTML:

```
<div class="app-layout">
  <div class="tweets">Tweets</div>
  <div class="replies">Replies</div>
  <div class="search">Search</div>
  <div class="messages">Messages</div>
</div>
```

Then we will apply some CSS to the `.app-layout` container element:

```
.app-layout {
  display: grid; /* 1 */
  grid-template-columns: 1fr 1fr 1fr 1fr; /* 2 */
  grid-template-rows: 100vh; /* 3 */
}
```

View a demo here

Here is the explanation of what we've done in the previous CSS:

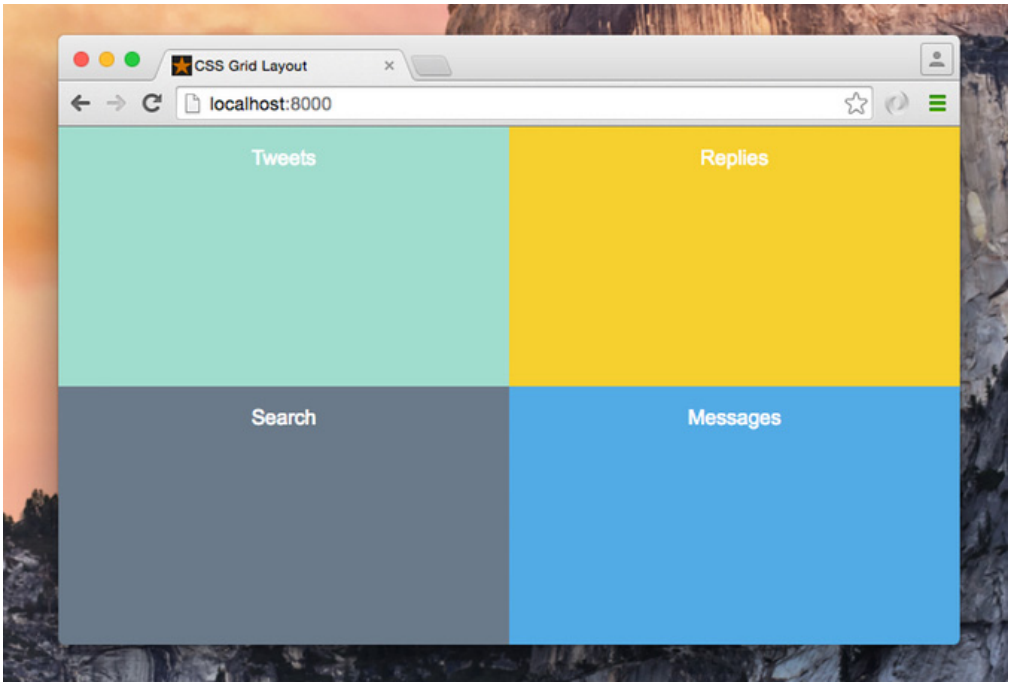
1. Set the display property to `grid`.
2. Divide the container element into four columns, each column is `1fr` (one fraction) of the free space within the grid container.
3. Create one row and set the height to be `100vh` (full viewport height).

As you can see, the Grid Layout Module adds a new value to the `display` property which is `grid`. The `grid` value is responsible for setting the `.app-layout` element to be a grid container, which also establishes a new grid formatting context for its contents. This property is required to start using Grid Layout.

The `grid-template-columns` property specifies the width of each grid column within the Grid, and in our case it divides the `.app-layout` container to four columns; each one is `1fr` (25%) of the available space.

The `grid-template-rows` specifies the height of each grid row, and in our example we only created one row at `100vh`.

A layout with two columns and two rows would look like this:



And we would use the following CSS:

```
.app-layout {
  display: grid;
  grid-template-columns: 1fr 1fr;
  grid-template-rows: 50vh 50vh;
}
```

View a demo here

We can also achieve the above example only on small screens by wrapping the code inside a media query. This opens up a great opportunity for us to customize the layout differently in different viewports. For example, we can create the previous layout only on viewports under `1024px` as follows:

```
@media screen and (max-width: 1024px) {
  .app-layout {
    display: grid;
```

```
    grid-template-columns: 1fr 1fr;
    grid-template-rows: 50vh 50vh;
  }
}
```

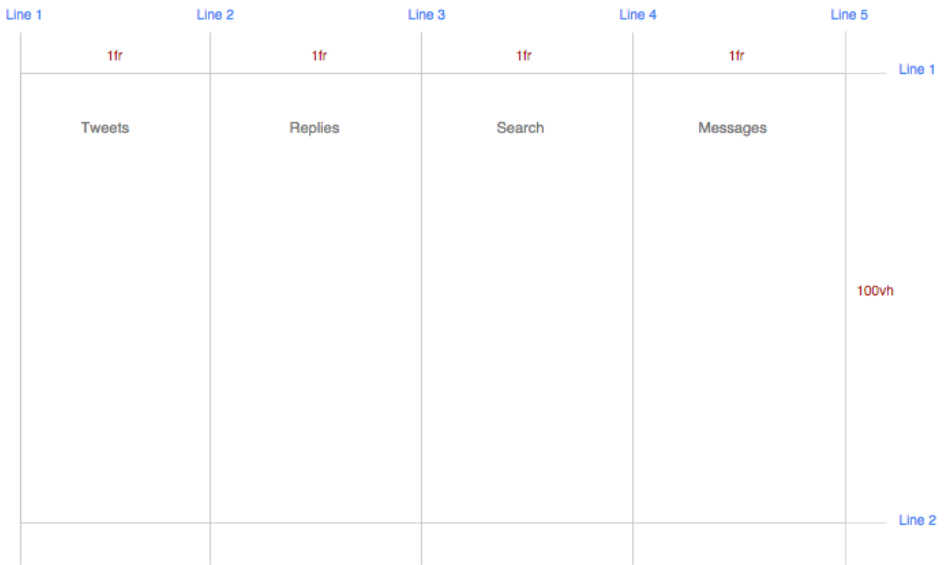
View a demo here

Grid Layout Module Concepts

Now that you've seen a simple example, there are some new concepts that I'd like to cover to give you a better understanding of Grid Layout. Although there are a lot of new concepts, I will only take a look at a few of them.

Grid Item
Grid items are the child elements of the grid container. In the above example, the `.tweets`, and `.replies` elements would qualify as grid items.

Grid Lines



A Grid Line is a line that exists on either side of a column or a row. There are two sets of grid lines: One set defining columns (the vertical axis), and another set defining rows (the horizontal axis).

From the above screenshot, which represents the first example, I've created four columns at `1fr` each, which will give us five vertical lines. I also created one row, which gives us two horizontal lines.

Let's see how we can position a grid item inside the grid container.

Position Items by Using a Line Number

You can refer to an exact line number in a grid by using the properties `grid-column-start` and `grid-column-end`. We then give these properties the start and end line numbers.

Looking at the previous example, this is how the browser positions the elements by default for us:

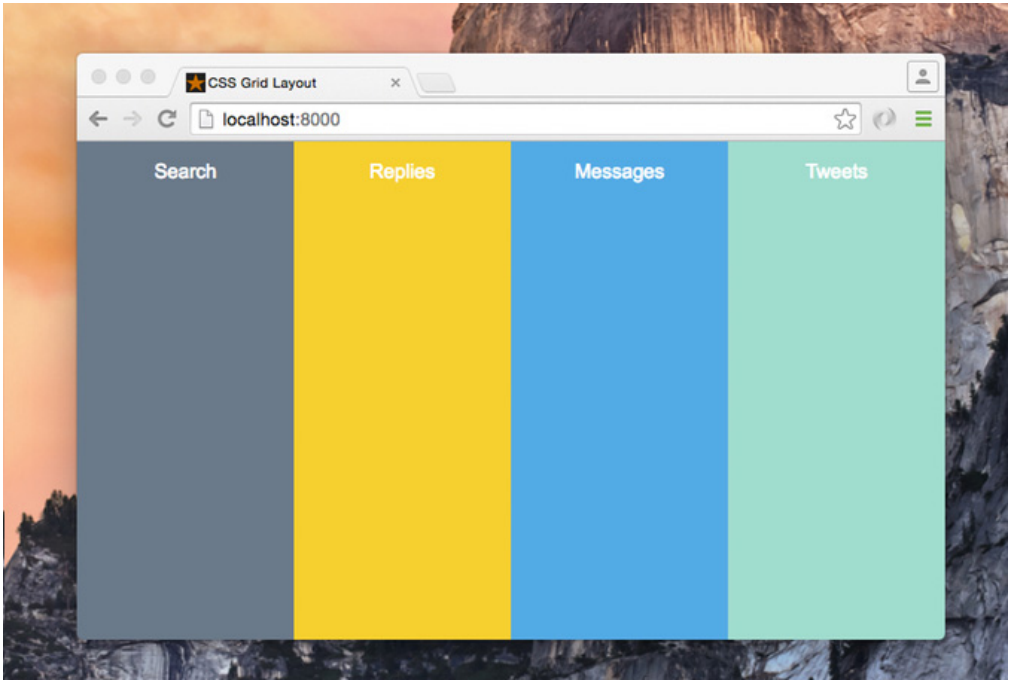
```
.tweets   {
  grid-column-start: 1;
  grid-column-end: 2;
  grid-row: 1;
}

.replies  {
  grid-column-start: 2;
  grid-column-end: 3;
  grid-row: 1;
}

.search   {
  grid-column-start: 3;
  grid-column-end: 4;
  grid-row: 1;
}

.messages {
```

```
  grid-column-start: 4;
  grid-column-end: 5;
  grid-row: 1;
}
```

Looking at the code for the `.tweet` column, this is what each of the three lines in the CSS does:

1. Position the child element starting from the first vertical line on the left.
2. End the element's position at the second vertical line.
3. Position the element inside the whole row.

You can change this by changing the order of elements with different positions, so the order of the elements will be: `.search`, `.replies`, `.messages`, and `.tweets`.



And we can do it as follows:

```
.tweets {
  grid-column-start: 4;
  grid-column-end: 5;
  grid-row: 1;
}

.replies {
  grid-column-start: 2;
  grid-column-end: 3;
  grid-row: 1;
}

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

.messages {
  grid-column-start: 3;
  grid-column-end: 4;
  grid-row: 1;
}
```

We can also use the `grid-column` shorthand property to set the start and end lines in one line:

```
.tweets {
  grid-column: 4 / 5;
  grid-row: 1;
}

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

.search {
```

```
  grid-column: 1 / 2;
  grid-row: 1;
}

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

View a demo here

This has changed the layout structure with only CSS while the markup is still as it was without any changes. This is a huge advantage of using the Grid Layout Module. We can rearrange the layout of elements independent of their source order, so we can achieve any desired layout for different screen sizes and orientations.

Position Items by Using Named Areas

A grid area is the logical space used to lay out one or more grid items. We can name a grid area explicitly using the `grid-template-areas` property, then we can place a grid item into a specific area using the `grid-area` property.

To make this concept more clear, let's redo the four-column example with the `search` column placed first:

```
.app-layout {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr 1fr;
  grid-template-rows: 100vh;
  grid-template-areas: "search replies messages tweets";
}
```

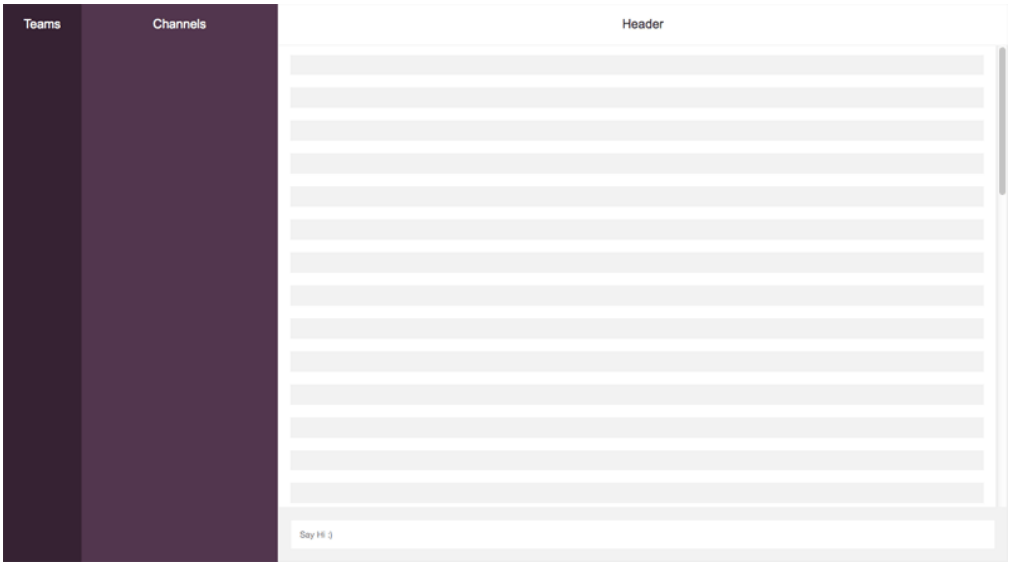
In the last line, we divide the grid container into four named grid areas, each name for a column. The next step is to position each grid item into a named area:

```
.search {
  grid-area: search;
}

.replies {
  grid-area: replies;
}

.messages {
  grid-area: messages;
}

.tweets {
  grid-area: tweets;
}
```

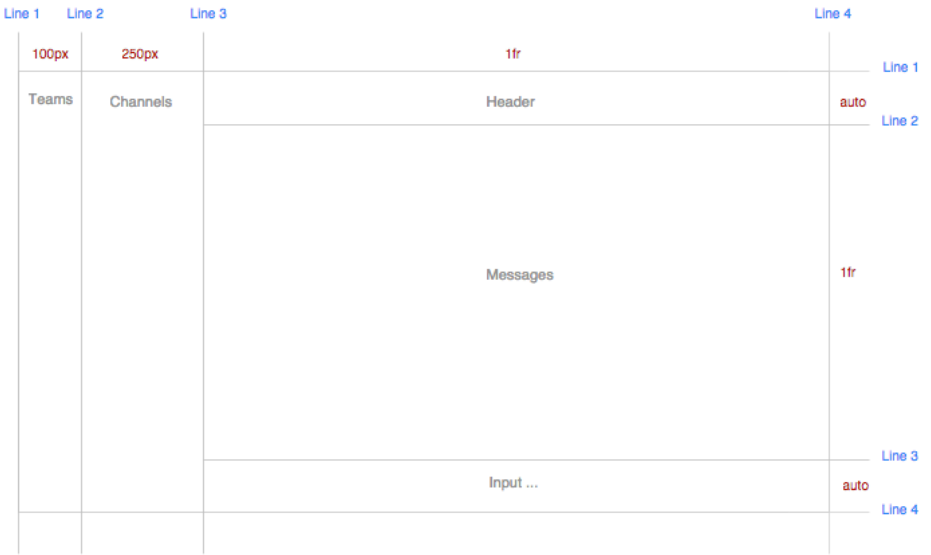
View a demo here

Slack Example

What about using the Grid Layout Module to implement a more complex example, for example, creating the building blocks of the Slack layout. Since we are talking about layouts, we will abstract and simplify the Slack design to the building blocks represented in the grid. Something like this:



From this layout we will create three vertical columns, and three horizontal rows, and we can visualize it using the grid lines as follows:



Here is the HTML:

```
<div class="app-layout">
  <div class="teams">Teams</div>
  <div class="channels">Channels</div>
  <div class="header">Header</div>
  <div class="messages">
    <ul class="message-list">
      <li></li>
      <li></li>
    </ul>
  </div>
  <div class="input">
 <input type="text" placeholder="CSS Grid Layout
↪ Module">
  </div>
</div>
```

And the CSS:

```
.app-layout {
  display: grid;
  height: 100vh;
  grid-template-columns: 100px 250px 1fr;
  grid-template-rows: auto 1fr auto;
}
```

Here I'm using the `grid-template-columns` property to create three columns at 100px, 250px, and the third column takes up the remaining available space. The last line creates three rows: The first and third rows with auto height while the middle row takes up the remaining available space.

The remainder of the CSS looks like this:

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

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

.header {
  grid-column: 3;
  grid-row: 1;
}

.messages {
  grid-column: 3;
  grid-row: 2;
}

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

View a demo here

We can also create the slack layout using named areas, which you can see in this demo.

Grid Layout Module vs Flexbox

Since many of you have started using Flexbox, you might wonder: When would it be appropriate to use Flexbox and when would it be more appropriate to use Grid Layout?

I found a good explanation from Tab Atkins:

> 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

Also, Rachel Andrew says:

> Grid Layout for the main page structure of rows and columns.
>
> Flexbox for navigation, UI elements, anything you could linearize.

CSS Grid Layout Module Resources

I have not covered all the Grid Layout concepts and syntax, so I recommend you check out the following resources to go deeper:

- CSS Grid Layout Module spec
- CSS Grid Layout Examples
- Grid by Example
- The future of layout with CSS: Grid Layouts
- Follow Rachel Andrew for updates and resources. She is doing a lot of great work in relation to Grid Layout.

Conclusion

As you've seen, the CSS Grid Layout Module is powerful because of its code brevity and the fact that you have the power to change the layout order without touching the markup. These features can help us permanently change the way we create layouts for the web.

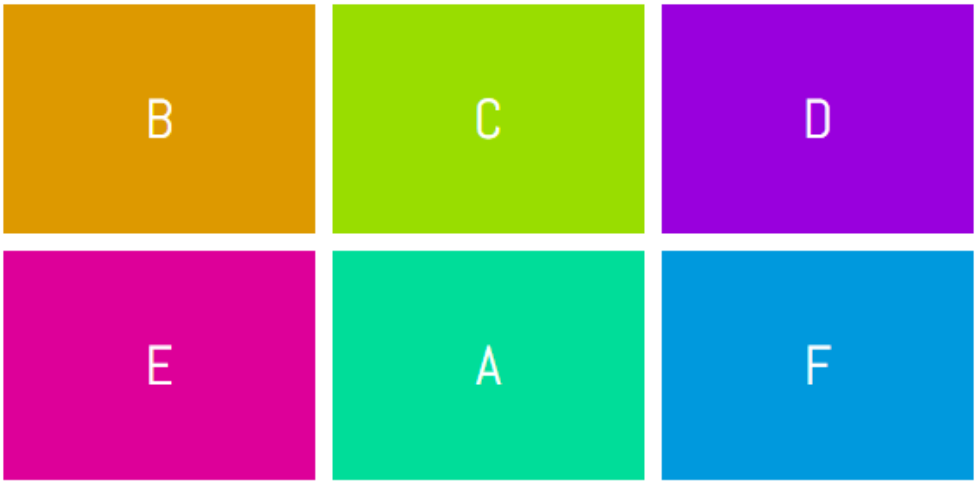
Chapter 2

Seven Ways You Can Place Elements Using CSS Grid Layout

by Nitish Kumar

In this chapter, the focus will be on specific ways in which you can lay out elements on the web using CSS Grid. Now, let's go over each one of them.

#1 Specifying Everything in Individual Properties



This is the version we have been using to place the elements in our previous articles. This method is verbose but easy to understand. Basically, the left/right and top/bottom bounds of an element are specified using `grid-column-start`/`grid-column-end` and `grid-row-start`/`grid-row-end` properties. If an element is only going to span one row or column, you can omit the `-end` properties, this way you will have to write a little less CSS.

In the demo below, element A has been placed in the second row and second column using the following CSS:

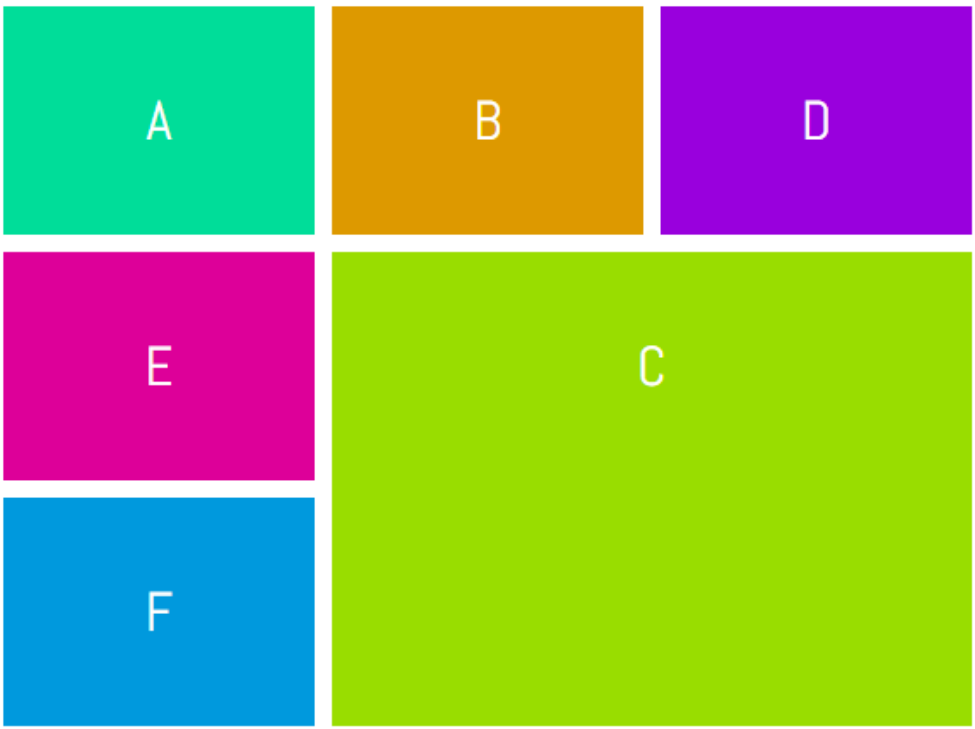
```
.a {
  grid-column-start: 2;
  grid-column-end: 3;
  grid-row-start: 2;
  grid-row-end: 3;
}
```

The same effect could be achieved by using:

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

See the demo Specifying Everything in individual Properties.

#2 Using `grid-row` and `grid-column`



Even though the CSS in our first example was readable and easy to understand, we had to use four different properties to place a single element. Instead of using four properties, we can just use two — `grid-column` and `grid-row`. Both these properties will take two values separated by a slash where the first value will determine the start line and the second value will determine the end line of our element.

Here is the syntax you need to use with these properties:

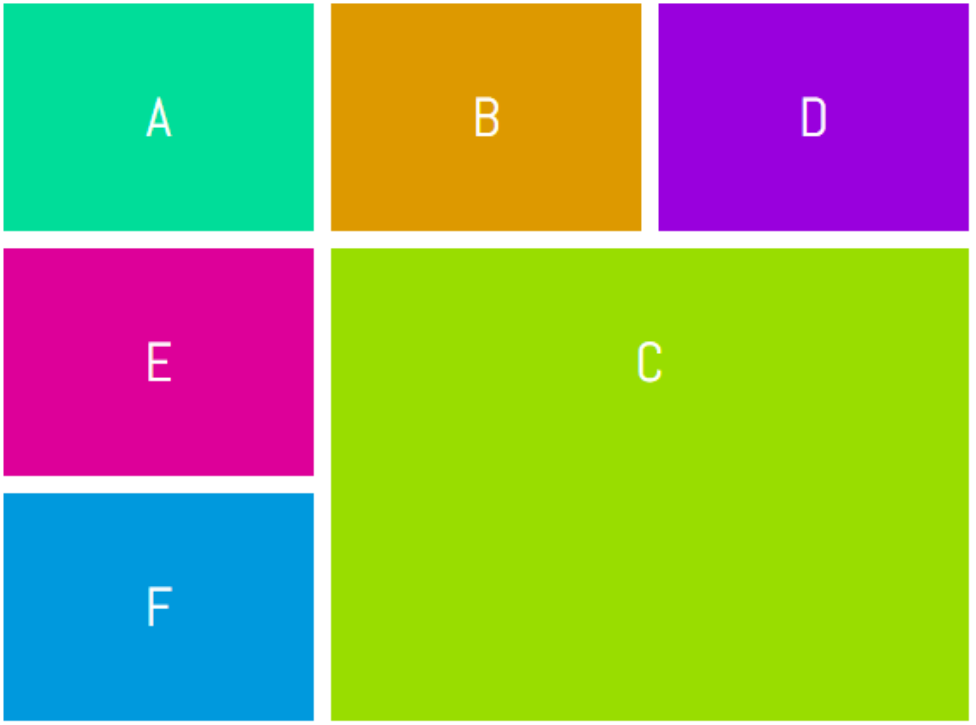
```
.selector {
  grid-row: row-start / row-end;
  grid-column: col-start / col-end;
}
```

To place item C in the bottom right corner of our grid, we can use the following CSS:

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

See the demo Using grid-row and grid-column.

#3 Using `grid-area`



Technically, the item we are laying out covers a specific area of the webpage. The boundary of that item is determined by the values we provide for the grid lines. All of these values can be supplied at once using the `grid-area` property.

This is what your CSS would look like when using this property:

```
.selector {
  grid-area: row-start / col-start / row-end / col-end;
}
```

If you have trouble remembering the correct order of these values, just keep in mind that first you have to specify the position of the top-left (`row-start` -

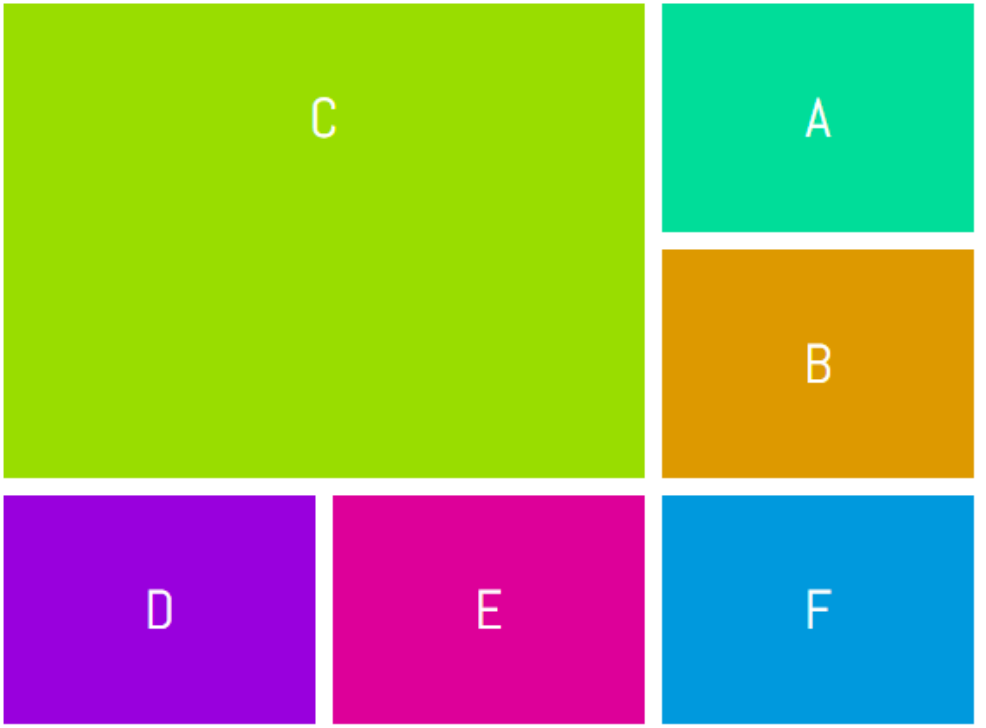
`col-start`) corner and then the bottom-right (`row-end` - `col-end`) corner of your element.

Just like the previous example, to place item C in the bottom right corner of our grid, we can use the following CSS:

```
.c {
  grid-area: 2 / 2 / 4 / 4;
}
```

See the demo Using grid-area.

#4 Using the span Keyword



Instead of specifying the end line while laying out elements, you can also use the `span` keyword to set the number of columns or rows a particular element will span.

This is the proper syntax when using the `span` keyword:

```
.selector {
  grid-row: row-start / span row-span-value;
  grid-column: col-start / span col-span-value;
}
```

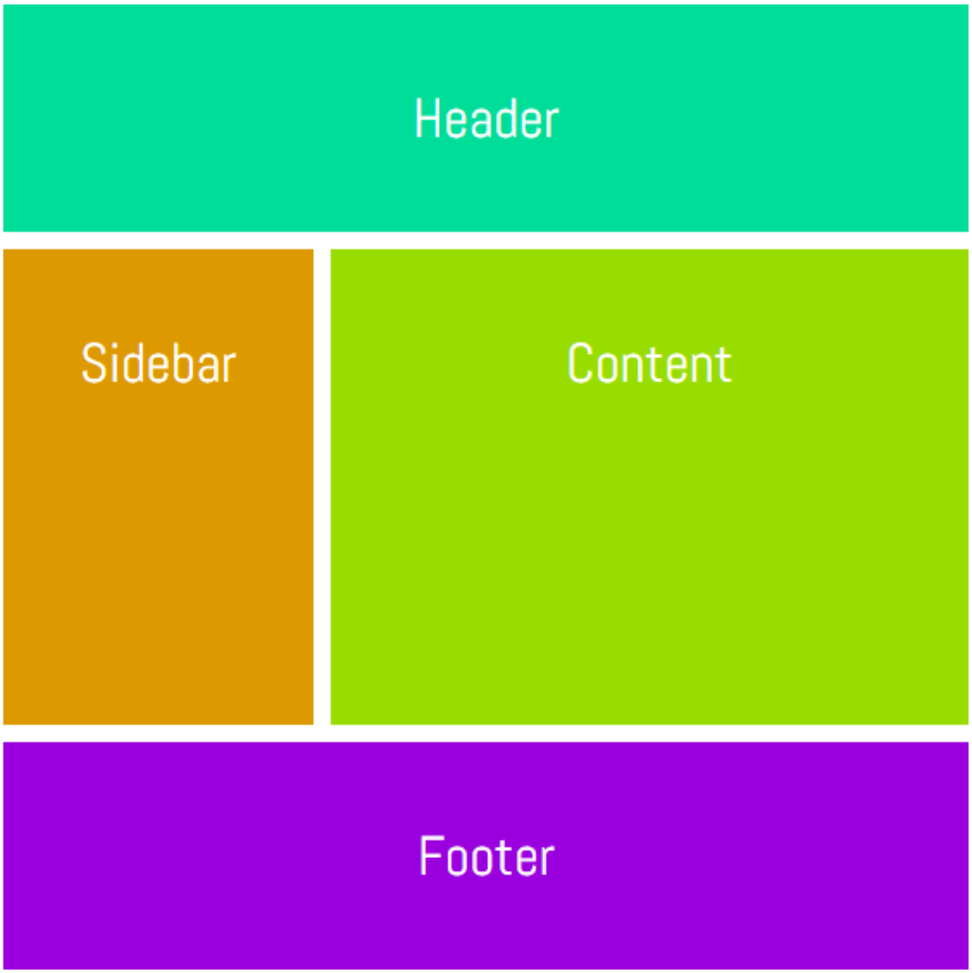
If your element spans across only one row or column you can omit both the `span` keyword and its value.

This time let's place item C in the top left corner of our grid. We can use the following CSS to do so.

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

See the demo Using span with grid lines.

#5 Using Named Lines



Until now, we have been using raw numbers to specify grid lines and it is easy to use when we are working with simple layouts. However, when you have to place several elements, it can get a bit confusing. Most of the times, an element on your page will fall under a specific category. For example, the header may go from column line `c1` to column line `c2` and from row line `r1` to row line `r2`. It would be a lot easier to properly name all the lines and then place your elements using these names instead of numbers.

Let's create a very basic layout to make the concept clearer. First, we will have to modify the CSS applied to our grid container:

```
.container {
  display: grid;
  grid-gap: 10px;
 grid-template-columns: [head-col-start] 180px
↪ [content-col-start] 180px [content-col-mid] 180px
↪ [head-col-end];
 grid-template-rows: [head-row-start] auto [head-row-end]
↪ auto [content-row-end] auto [footer-row-end];
}
```

What I have done above is assign names to all the lines based on the type of content that they will enclose. The idea here is to use names that provide us with some insight into the placement of different elements. In this particular example, our header element spans across all the columns. Therefore, assigning the name "head-col-start" and "head-col-end" to the first and last column line respectively will make it clear that these lines represent the left and right end of our header. All other lines can be named in a similar fashion. After all the lines have been named, we can use the following CSS to place all our elements.

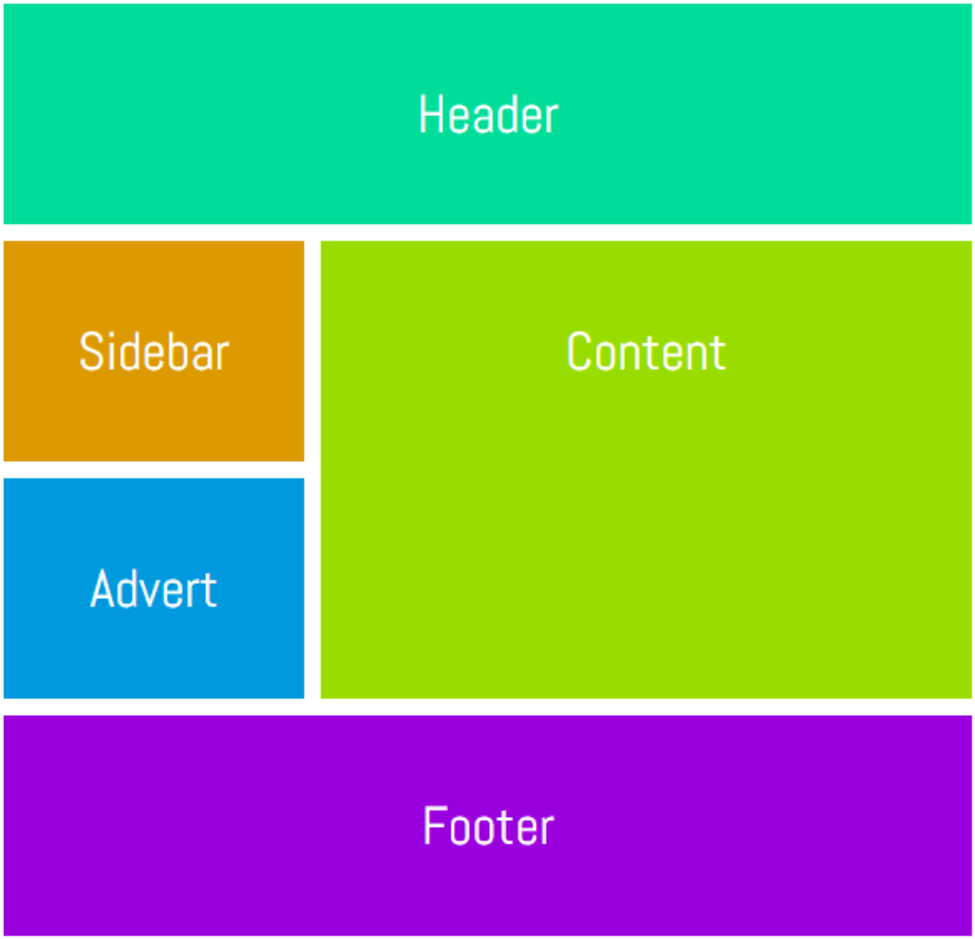
```
.header {
  grid-column: head-col-start / head-col-end;
  grid-row: head-row-start / head-row-end;
}

.sidebar {
  grid-column: head-col-start / content-col-start;
  grid-row: head-row-end / content-row-end;
}

.content {
  grid-column: content-col-start / head-col-end;
  grid-row: head-row-end / content-row-end;
}
```

```
.footer {
  grid-column: head-col-start / head-col-end;
  grid-row: content-row-end / footer-row-end;
}
```

Although we had to write more CSS than usual, now just looking at the CSS will give us an idea of where an element is located.

See the demo Using Named Lines.

#6 Using Named Lines with a Common Name and the span Keyword



In the previous method, all lines had different names marking the starting, midpoint or the end of an element. For example, "content-col-start" and "content-col-mid" marked the starting and mid point of the content section of our webpage. If the content section covered a few more rows, we would have to come up with additional line names like "content-col-mid-one", "content-col-mid-two" and so on.

In situations like these, we can just use a common name like "content" for all grid lines of our content section and then use the `span` keyword to specify how many of those lines an element spans. We can also just mention a number along with the line name to set the number of rows or columns an element would span.

Using this method, the CSS would look like this:

```
.selector {
 grid-row: row-name row-start-number/ row-name
↪ row-end-number;
 grid-column: col-name col-start-number / span col-name
↪ col-to-span;
}
```

Like the last method, this one also requires you to modify the CSS of your grid container.

```
.container {
  display: grid;
  grid-gap: 10px;
 grid-template-columns: [one-eighty] 180px [one-eighty] 180px
↪ [one-eighty] 180px;
 grid-template-rows: [head-row] auto [content-row] auto
↪ [content-row] auto [content-row] auto [footer-row] auto;
}
```

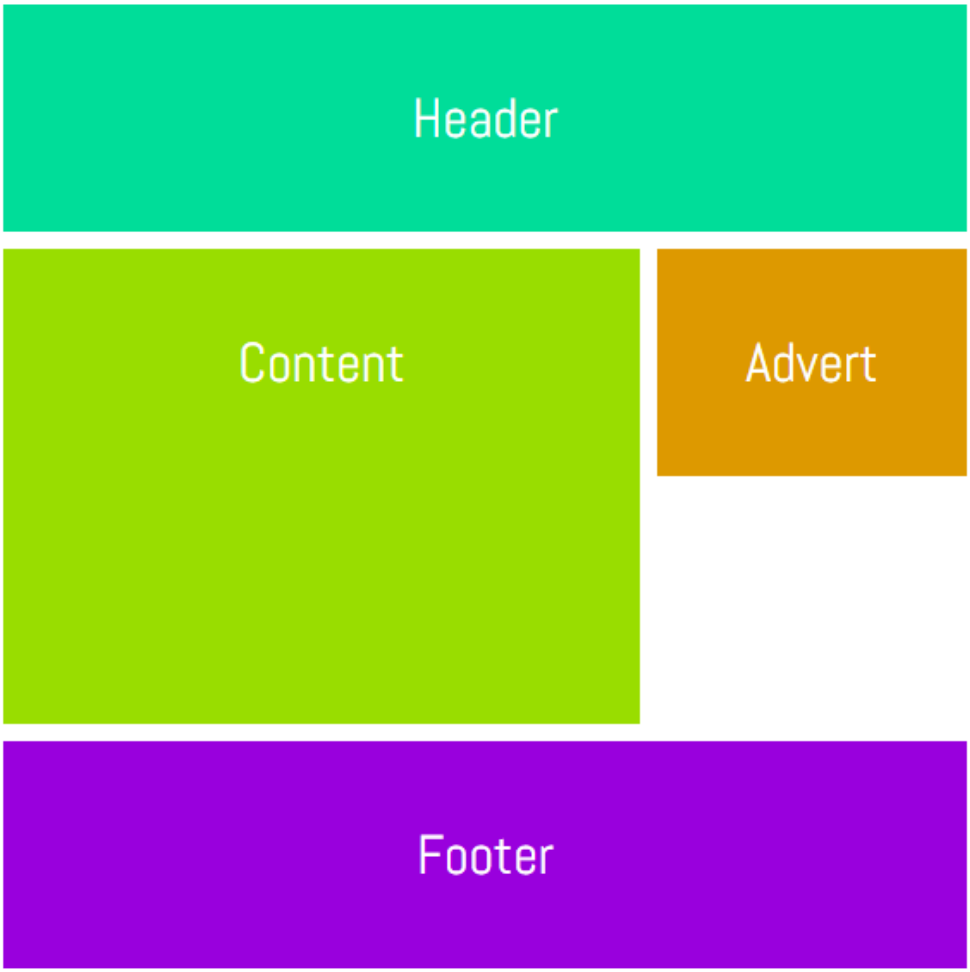
Each of the named column lines has the same name representing their width in pixels and each named row line represents the rows covered by a specific section of the webpage. In this demo, I have introduced an advertisement section just under the sidebar. Here is the CSS:

```
.header {
  grid-column: one-eighty 1 / one-eighty 4;
  grid-row: head-row / content-row 1;
}
```

```
.sidebar {
  grid-column: one-eighty 1 / one-eighty 2;
  grid-row: content-row 1 / content-row 2;
}

.advert {
  grid-column: one-eighty 1 / one-eighty 2;
  grid-row: content-row 2 / content-row 3;
}

.content {
  grid-column: one-eighty 2 / one-eighty 4;
  grid-row: content-row 1 / span content-row 2;
}

.footer {
  grid-column: one-eighty 1 / span one-eighty 3;
  grid-row: content-row 3 / footer-row;
}
```

See the demo Using Named Lines with Span.

#7 Using Named Grid Areas



Instead of using lines, we can also place elements by assigning names to different areas. Again, we will have to make some changes to the CSS of our grid container.

The CSS for our container should now look like this:

```
.wrapper {
  display: grid;
  grid-gap: 10px;
  grid-template-columns: 180px  180px  180px;
```

```
  grid-template-areas:  "header  header   header"
                        "content content  advert"
                        "content content  ......"
                        "footer  footer   footer";
}
```

A single dot (`.`) or a sequence of dots will create an empty cell with nothing in it. All the strings need to have the same number of columns. That's why we had to add the dots instead of leaving it completely blank. For now, the named grid area can only be rectangular. However, this may change in future versions of the spec. Let's take a look at the CSS of all our elements.

```
.header {
  grid-area: header;
}

.content {
  grid-area: content;
}

.advert {
  grid-area: advert;
}

.footer {
  grid-area: footer;
}
```

Once you have defined all the grid areas, assigning them to various elements is pretty straightforward. Keep in mind that you can't use any special characters while assigning names to areas. Doing so will make the declaration invalid.

See the demo Using Named Grid Areas.

Chapter 3

How to Order and Align Items in Grid Layout

by Nitish Kumar

In this tutorial, you will learn how to control the order in which items are placed using the Grid Layout module. After that, we will discuss how to control the alignment of different items in the grid.

How the Order Property Works in Grid Layout

The `order` property can be used to specify the order in which different items should be placed inside a grid. By default, the items are placed in the order in which they appear in the DOM. For example, if item A is above item B in the

actual source document, it will also be placed in the grid before item B. Depending on your project, this may or may not be the desired behavior.

The `order` property can be very useful, especially when there are lots of items or the items are being added dynamically. In such cases, if you want an item to be placed always at the end of the grid, you can do so easily using the `order` property.

Items with the **lowest order value are placed first** in the grid. Items with **higher values are placed later**. Items which have **the same order value will be placed in the order in which they appear in the source document**.

Let's take a look at an example.

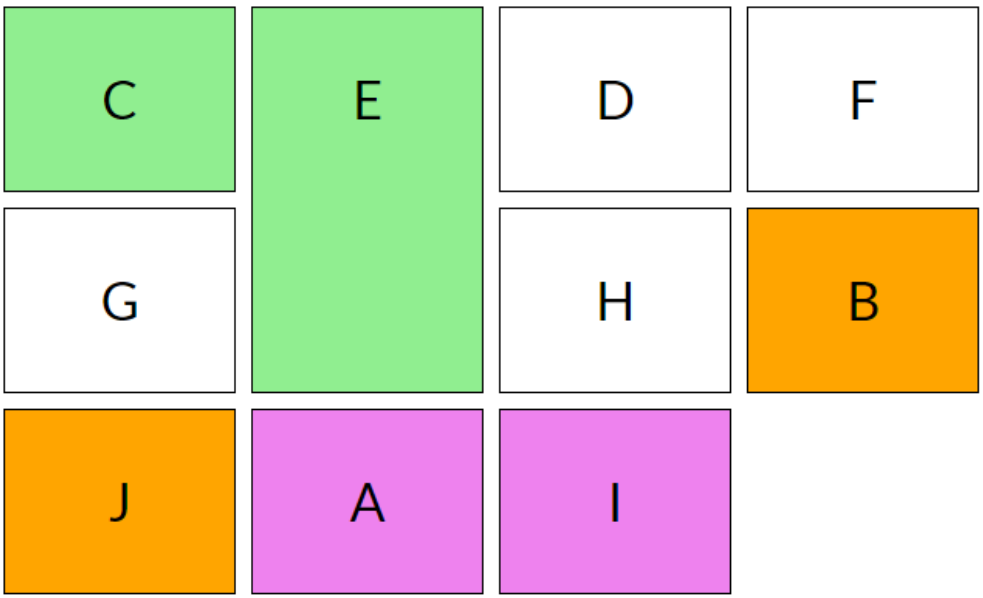
This is our markup:

```
<div class="container">
  <div class="item a">A</div>
  <div class="item b">B</div>
  <div class="item c">C</div>
  <div class="item d">D</div>
  <div class="item e">E</div>
  <div class="item f">F</div>
  <div class="item g">G</div>
  <div class="item h">H</div>
  <div class="item i">I</div>
  <div class="item j">J</div>
</div>
```

Here is the CSS for placing the grid items:

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

.e {
  grid-row-start: 1;
```

```
  grid-row-end: 3;
}

.b, .j {
  order: 2;
}

.a, .i {
  order: 3;
}
```



If you recall the steps from the auto-placement algorithm tutorial, you know that the algorithm will place the items with an explicitly specified row position before placing items with no definite position. So, even though item D, without a definite row or column position, comes before item E in the actual document, it will still be placed after item E (which has a definite row position: `grid-row-start: 1` and `grid-row-end: 3`).

Among the items with no definite position, items with the lowest `order` value will be placed first. That's why items D, F, G and H are placed before items A and

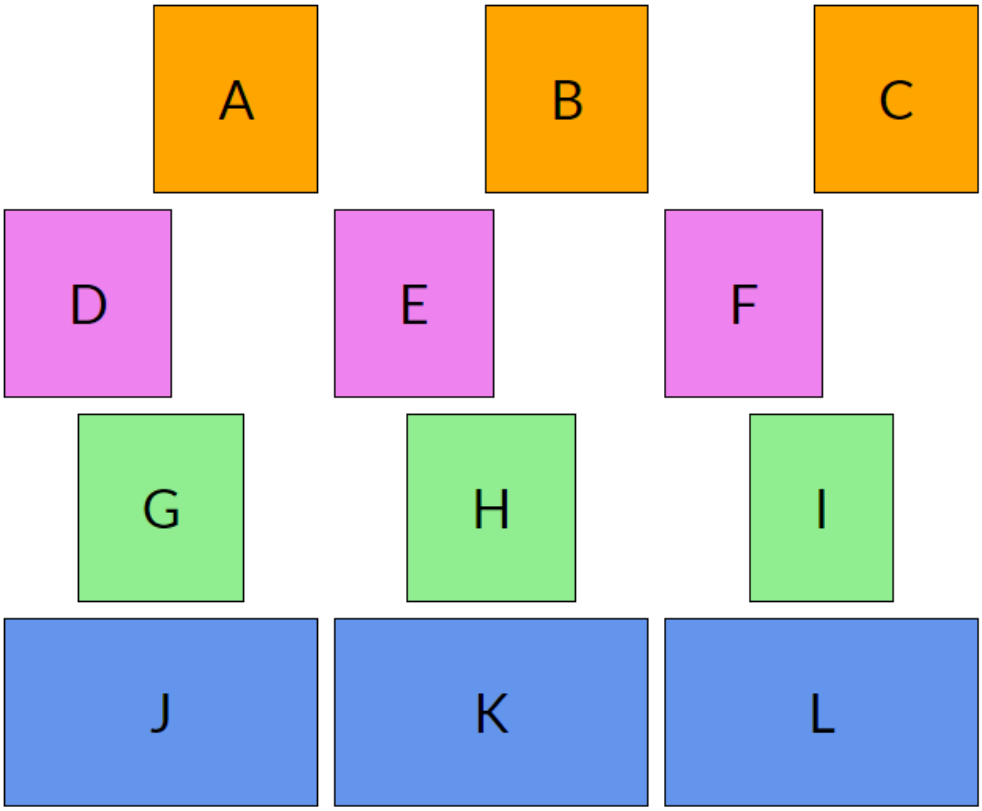
B. B and J have the same `order` value, therefore they are placed in the same order in which they appear in the document. Please note that both B and J are still placed before placing A and I because they have a lower `order` value.

See the demo The order Property in Grid Layout.

You need to keep **accessibility** in mind before reordering grid items using the `order` property. This property does not affect ordering of items in non-visual media. It also does not affect the default traversal order when navigating the document using sequential navigation modes like tabs. Therefore, don't use this property unless the visual order of elements needs to be out-of-sync with the speech and navigation order.

Performing logical reordering of the grid items using the order property will make the style sheets non-conforming.

Aligning Content Along the Row Axis in Grid Layout



You can control the alignment of different grid items along the row axis using `justify-self` and `justify-items` .

The `justify-self` property aligns the content of a single grid item while `justify-items` aligns all the items in the grid.

The `justify-self` property can have four possible values:

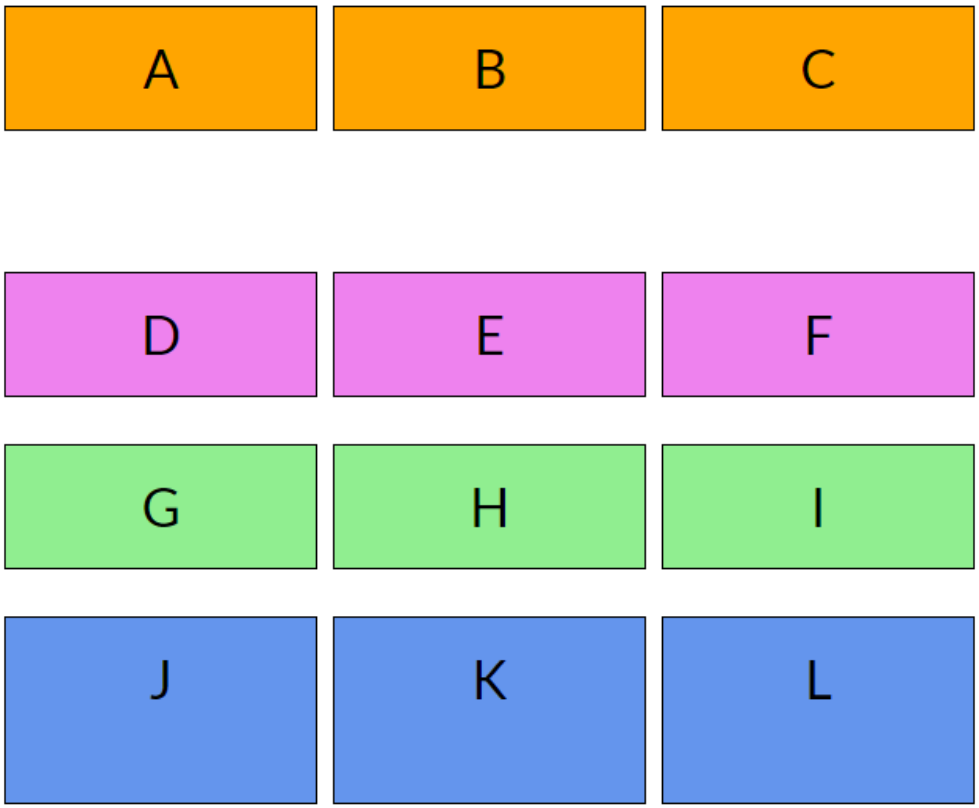
- `end` aligns content to the right of the grid area
- `start` aligns content to the left of the grid area
- `center` aligns content to the center of the grid area
- `stretch` fills the whole width of the grid area

The default value of `justify-self` is `stretch`.

The `justify-items` property also aligns items with respect to the row axis and takes the same four values as the `justify-self` property. The default value is also `stretch`.

See the demo Aligning Content Along Row Axis.

Aligning Content Along the Column Axis in Grid Layout



You can also align content along the column axis using `align-self` for single grid items and `align-items` for all the items in the grid.

Just like the previous two properties, both `align-self` and `align-items` can have four possible values: `start`, `end`, `center` and `stretch`.

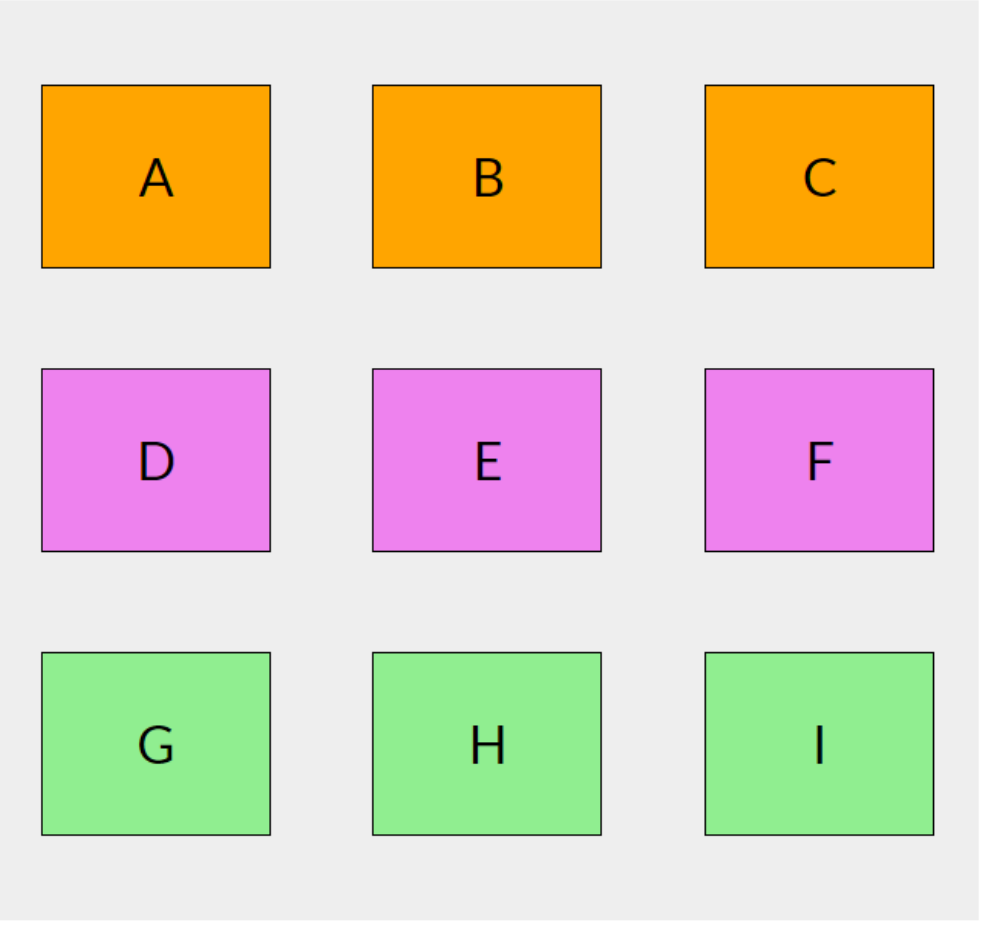
The `start` and `end` values align the content to the top and bottom of the grid area respectively. The `center` value aligns the content to the center of the grid area and `justify` fills the whole height of the grid area.

The extra space between the first and second row in the demo occurs because the items in the first row are aligned to the top and the items in the second row are aligned to the bottom.

Below is `align-self` to align content along the column axis in action.

See the demo Aligning Content Along Column Axis.

Aligning the Whole Grid



Sometimes, the outer edges of the grid may not align with the outer edges of the container. This can happen when the grid rows or columns have a fixed size. In such cases, you may want to align the grid in a specific direction.

Just like the previous properties, alignment along the row axis can be achieved using the `justify-content` property and alignment along the column axis can be achieved using the `align-content` property.

Both these properties are applied to the container grid and besides the usual `start`, `end`, `center` and `stretch` values, they also accept `space-around`, `space-between` and `space-evenly` as valid values.

- `space-around` places an equal amount of space between each grid track and half of that space between the outer edges of the grid and its container
- `space-between` places an equal amount of space between each grid track and no space between the outer edges of the grid and its container
- `space-evenly` places an equal amount of space between each grid track as well as the outer edges of the grid and its container.

The default value of both `justify-content` and `align-content` is `start`.

I have used the CSS below to align the grid items in the following demo:

```
.container {
  justify-content: space-around;
  align-content: space-evenly;
}
```

See the demo Aligning the Whole Grid.

Aligning items this way can also have some unintended side effects like adding extra width or height to grid items that span more than one column or row respectively. The amount of width and height that gets added to a grid item in this case depends on the number of gutters that the item crosses in each respective direction.

Conclusion

In this tutorial, we have covered all the basics of ordering and aligning items in Grid, which can help you to gain precise control over your Grid-based layouts.

Chapter 4

A Step by Step Guide to the Auto-Placement Algorithm in CSS Grid

by Nitish Kumar

In this tutorial, I'll be going over all the steps the **auto-placement algorithm of the CSS Grid Layout module** follows when positioning elements. These steps are controlled by the `grid-auto-flow` property.

In Seven Ways You Can Place Elements Using CSS Grid Layout, I explained all the different ways Grid lets you arrange elements on the web. However, in my previous articles I explicitly specified the position of just one element in the grid. As for the rest of the items, they got placed properly based on an algorithm.

Here, I am going to show you how this algorithm works. This way, the next time an element ends up in an unexpected location, you are not left scratching your head wondering what just happened.

Basic Concepts for a Better Grasp of the Auto-placement Algorithm

Let's go over some fundamental concepts before diving into the workings of the algorithm.

- **Anonymous grid items** - If you place some text directly inside a grid container without wrapping it in any tag, it will form its own anonymous grid item. You can't style an anonymous grid item because there is no element for you to style, but it still inherits style rules from its parent container. On the other hand, note that white space inside the grid container will not create its own anonymous grid item
- **Value of grid spans** - Unlike grid positions, the algorithm has no special rules to resolve the value of grid spans. If not explicitly specified, their value is set to 1 (the item only occupies its own cell)
- **Implicit grid** - The grid built on the basis of the value of properties like `grid-template-rows`, `grid-template-columns` and `grid-template-areas` is called **explicit grid**. Now, if you specify the position of a grid item in such a way that it lies outside the bounds of the explicit grid, the browser will generate additional grid lines to accommodate the item. These lines, along with the explicit grid, form the implicit grid. You can read more about it in Where Things Are at in the CSS Grid Layout Working Draft. The auto-placement algorithm can also result in the creation of additional rows or columns in the implicit grid.

Finally, I'd like to make the following preliminary point. The default value of the `grid-auto-flow` property, which is the property controlling the algorithm, is `row`. This is also the value I am going to assume in the following explanation of the auto-placement algorithm. On the other hand, if you explicitly set the above property to `column`, remember to replace instances of the term *row* with the term *column* in my explanation of the algorithm. For example, the step "Placement of Elements With a Set Row Position but No Set Column Position" will become "Placement of Elements With a Set Column Position but No Set Row Position".

Now, enable the experimental features flag in your favorite modern browser to follow along and let's go over the details of all the steps the algorithm follows to build the layout.

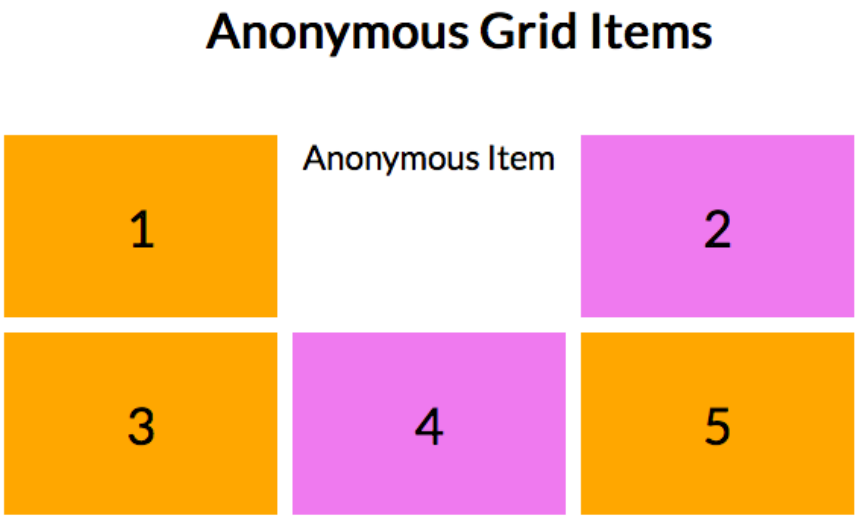
Step #1: Generation of Anonymous Grid Items

The first thing that happens when the algorithm is trying to place all the items inside a grid is the creation of anonymous grid items. As I mentioned earlier, you cannot style these elements because there is no item to apply the style to.

The markup below generates an anonymous grid item from the inter-element text:

```
<div class="container">
  <span class="nonan">1</span>
    Anonymous Item
  <div class="nonan floating">2</div>
  <div class="nonan">3</div>
  <div class="nonan floating">4</div>
  <div class="nonan">5</div>
</div>
```

Besides the generation of an anonymous item, one more thing to notice is that the grid placement algorithm ignores the CSS floats applied to *div 2* and *div 4* .



See the demo Anonymous Grid Items.

Step #2: Placement of Elements with an Explicitly Specified Position

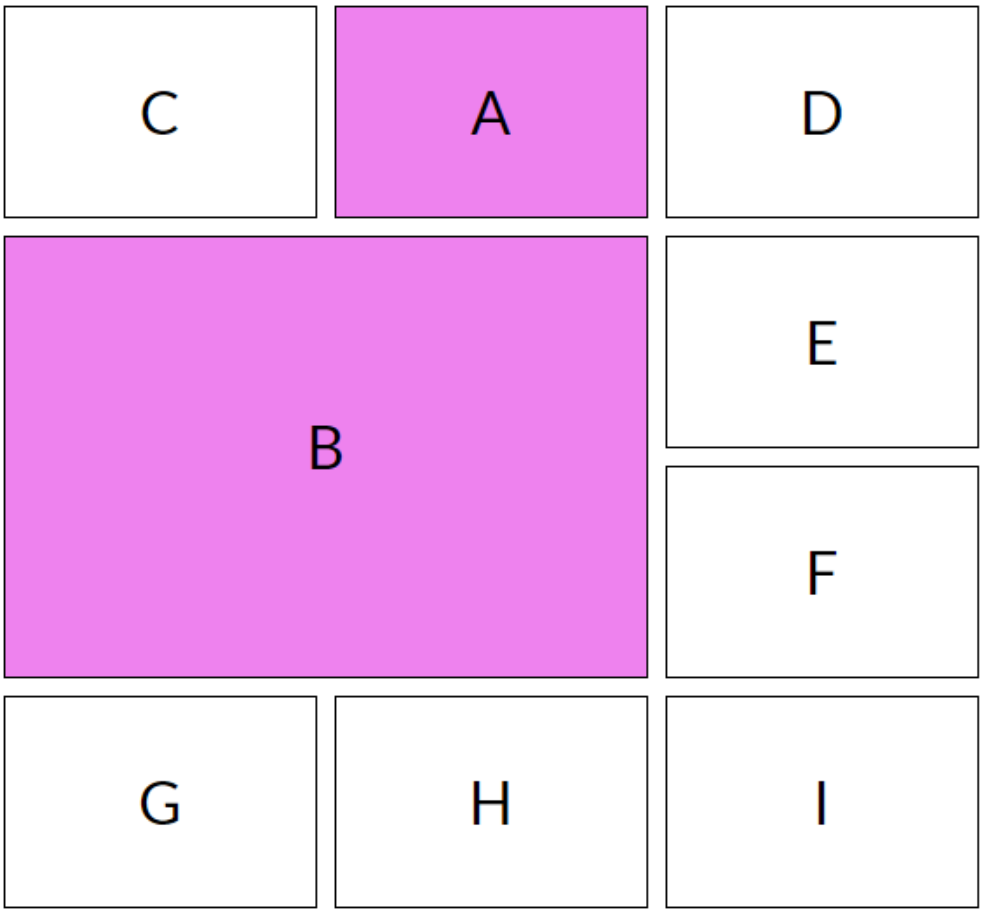
For this and the next few steps, I will be using a grid of nine different items to show how they are going to be placed.

Here's the relevant markup:

```
<div class="container">
  <div class="item a">A</div>
  <div class="item b">B</div>
  <div class="item c">C</div>
  <div class="item d">D</div>
  <div class="item e">E</div>
  <div class="item f">F</div>
  <div class="item f">G</div>
  <div class="item f">H</div>
  <div class="item f">I</div>
</div>
```

The first items to be placed in the grid are those with an **explicitly set position**, which in the example below are items A and B. Just ignore all other items in the grid for now. Here is the CSS that explicitly sets the position of A and B:

```
.a {
  grid-area: 1 / 2 / 2 / 3;
}

.b {
  grid-area: 2 / 1 / 4 / 3;
}
```



The algorithm positions items A and B according to the values of their respective `grid-area` property. In particular, the algorithm:

- Sets the position of the top left corner of both A and B using the first and second value of the `grid-area` property
- Sets the position of the bottom right corner of both A and B using the third and fourth value of the`grid-area` property.

See the demo Placing Elements with Explicit Positions.

Step #3: Placement of Elements With a Set Row Position but No Set Column Position

Next, the algorithm places the elements whose **row position** has been **set explicitly** by using the `grid-row-start` and `grid-row-end` properties.

For this example, I will be setting only the grid-row values of item C and item D to give them a definite row position. Here is the CSS for placing both of them:

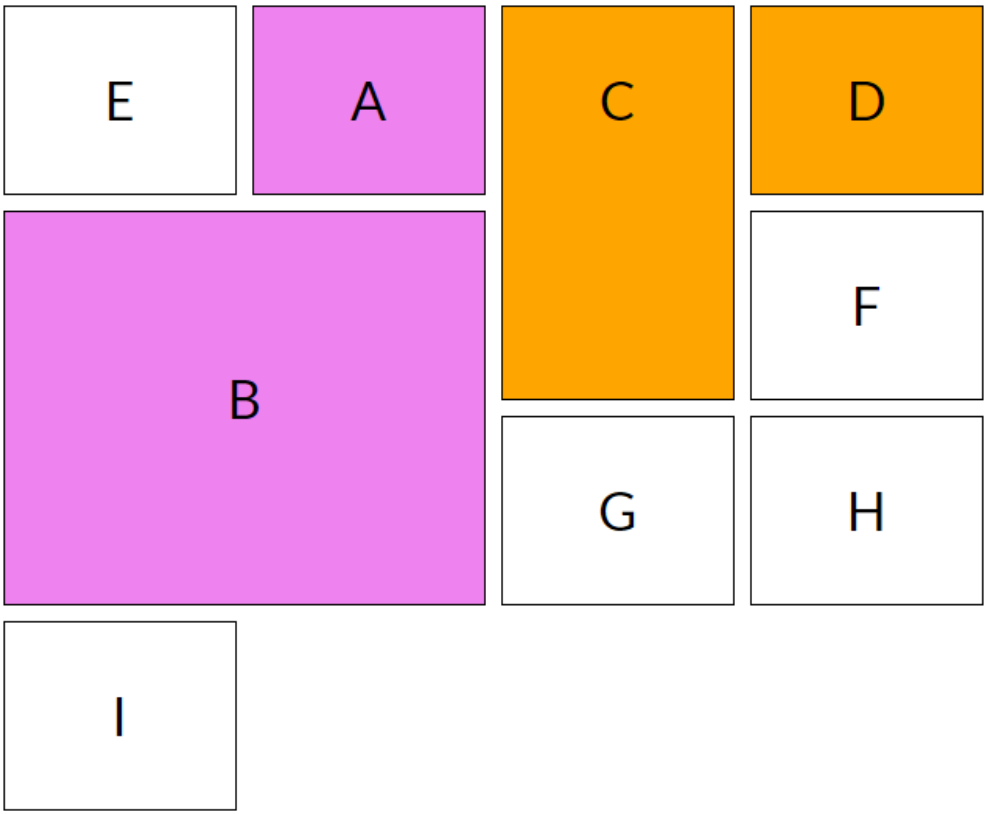
```
.c {
  grid-row-start: 1;
  grid-row-end: 3;
}

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

To determine the column position, which is not explicitly set, the algorithm behaves in one of two ways, according to the **packing mode**:

- sparse packing (default)
- dense packing

Sparse Packing in Step #3

This is the default behavior. The column-start line of our item will be set to the smallest possible line index which ensures that there won't be any overlap between the item's own grid area and the cells already occupied by other items. The column-start line also needs to be past any other item already placed in this row *by this step*. Please note that I wrote *by* this step and not *until* this step.



To clarify this point further, item D in the demo below did not move to the left of item A, even though it could fit in there without any overlap. This is due to the fact that the algorithm does not allow any item with an **explicitly set row position** but **not set column position** to be placed **before any other similarly positioned item in that specific row** (item C in this case). In fact, if you remove the grid-row rules applied to item C, then item D will move to the left of item A.

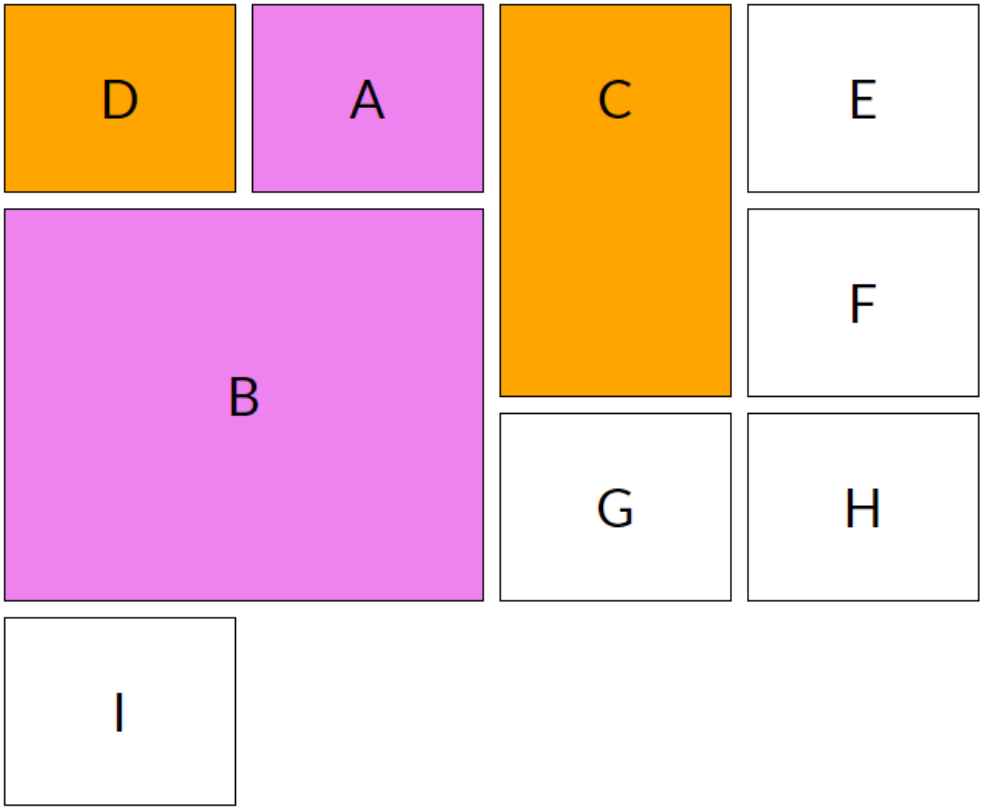
In short, item D, which has a definite row position but no explicitly set column position, can end up being placed before item A, but only if C does not interfere (interference takes place in this case because C, like D, has a definite row position but no set column position and is in the same row as D).

See the demo Placing Elements with Definite Grid Positions.

Dense Packing in Step #3

If you want to fill that empty space before item A with item D, you will have to set the value of the `grid-auto-flow` property to `row dense`.

```
.container {
  grid-auto-flow: row dense;
}
```



In this case too, the column-start line is placed at the smallest index which does not cause any overlap with other grid items. The only difference is that this time, if there is some empty space in a row where our element can fit without any overlap, it will be placed in that position, without considering the previous item in the same row with the same position rules (in this case item C).

See the demo Elements with Definite Grid Positions - Dense.

Step #4: Determining the Number of Columns in the Implicit Grid

Next, the algorithm tries to determine the number of columns in the implicit grid. This is done by following a series of steps:

- The algorithm starts with the number of columns in the explicit grid.

- Then it goes through all the grid items with a definite column position and adds columns to the beginning and end of the implicit grid to accommodate all these items.
- Finally, it goes through all the grid items without a definite column position. If the biggest column span among these items is greater than the width of the implicit grid, it adds columns at the end of the grid to accommodate that column span.

Step #5: Placement of Remaining Items

At this point, the algorithm has placed all the items whose position has been explicitly specified as well as items whose row positions are known. Now, it will position all other remaining items in the grid.

Before we go into more detail, you should know about the term **auto-placement cursor**. It defines the current insertion point in the grid, specified as a pair of row and column grid lines. To start off, the auto-placement cursor is placed at the start-most row and column in the implicit grid.

Once more, the packing mode, determined by the value of the `grid-auto-flow` property, controls how these items are positioned.

Sparse Packing in Step #5

By default, the remaining items are placed **sparsely**. Here is the process the algorithm follows to place them.

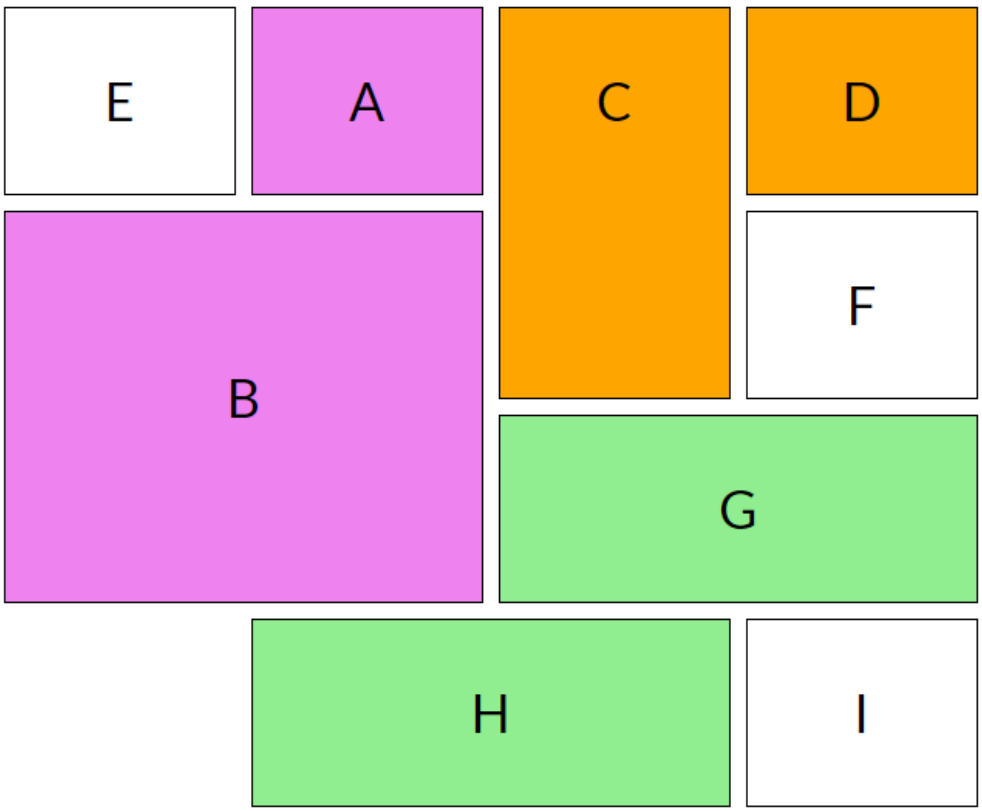
If the item has no definite position in either axis:

- The algorithm increments the cursor's column position until: a) either there is no overlap between already placed items and the current item b) or the cursor's column position plus the item's column span overflow the number of columns in the implicit grid.
- If such a non-overlapping position is found, the algorithm sets the item's `row-start` and `column-start` values to the cursor's current position. Otherwise, it increases the row-position by 1, sets `column-start` to the start-most line in the implicit grid and repeats the previous step.

If the item has a definite column position:

- The column position of the auto placement cursor is set to the item's `column-start` line. If the value of this new position is less than the cursor's previous column position, the row position is increased by one.
- Next, the row position is increased by one until the algorithm reaches a value where the grid item does not overlap with any of the occupied grid cells. If needed, extra rows can be added to the implicit grid. Now the item's row start line is set to the cursor's row position and the item's row end line is set according to its span.

Let's clarify the steps listed above with what's happening in the following demo.



Placement of Items E and F without Definite Position in Either Axis

When the algorithm deals with item **E**, which has **no explicitly set column or row position**, the auto-placement cursor is set to row 1 and column 1. Item E occupies just one cell and it can fit in the top left corner without any overlap. Therefore, the algorithm simply places **item E at position row 1 / column 1**.

The next item without an explicitly set column or row position is **F**. At this point, the column position for the auto-placement cursor is increased to 2. However, the position row 1 / column 2 is already occupied by item A. This means that the algorithm will have to keep increasing the column position. This goes on until the auto-placement cursor reaches column 4. Since there are no more columns, the row position for the auto-placement cursor is increased by 1 and the column position is set to 1. Now the row position is 2 and the column position is 1. The algorithm again starts increasing the column position by 1 until it reaches column 4. The space at **row 2 and column 4 is currently empty and can be occupied by item F**. The algorithm places item F there and moves on to the next item.

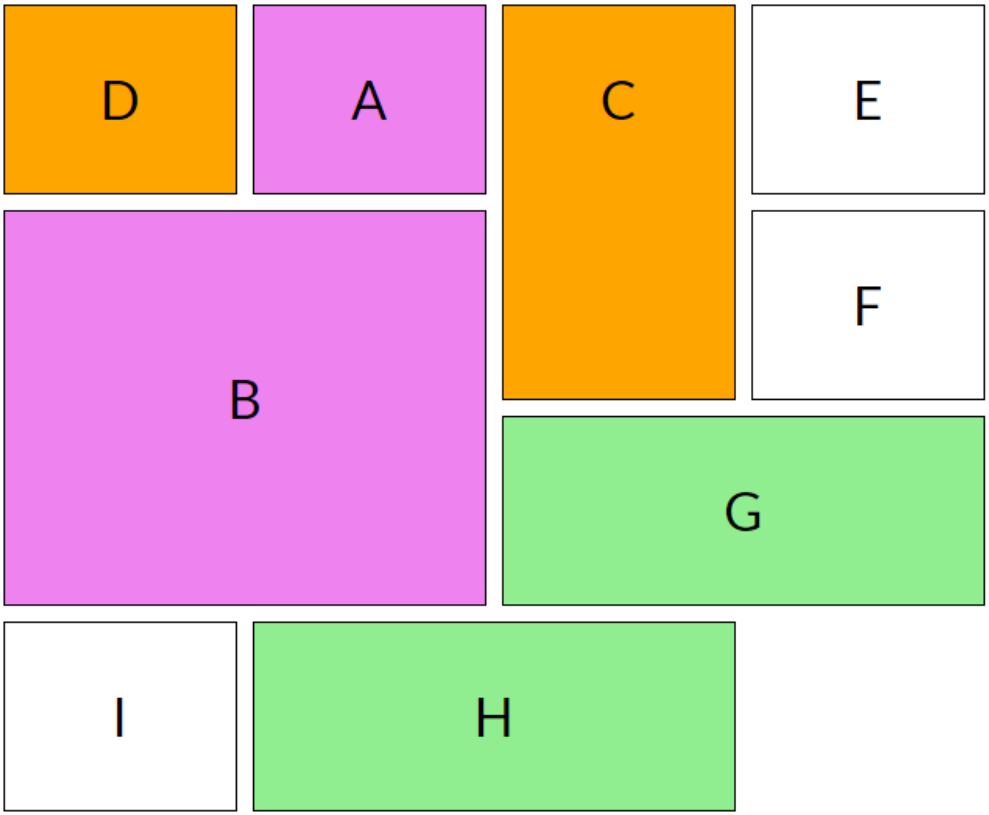
Placement of Items G and H with a Definite Column Position

The remaining items with a **definite column position** in our demo are G and H. Let's examine **G**. The column position of the auto-placement cursor is set to be equal to the value of the `grid-column-start` property for item G, which is 3. Since this new value is less than the previous column value (which was 4), the row position is increased by one. The current row and column positions are now 3 and 3 respectively. The space at **row 3 and column 3** is currently empty and item G can be placed there without any overlap. Therefore, the algorithm places item G there. The same steps are then repeated for the placement of item H.

See the demo Positioning Remaining Items by SitePoint (@SitePoint).

Dense Packing in Step #5

Things are handled a bit differently when the `grid-auto-flow` property is set to `row dense`. When placing a grid item with no definite position, the cursor's current position is set to the start-most row and column line in the implicit grid **before** the item's position is determined.



Unlike the previous example, item I is placed to the left of item H because the cursor position gets **reset to the start-most row and column line** in the implicit grid **instead of starting from the last item placed**. At this point, while looking for a suitable position to place item I without any overlap, the cursor finds the spot to the left of item H and places it there.

See the demo Positioning Remaining Items - Dense.

Conclusion

In this tutorial, I went over all the steps followed by the auto-placement algorithm of the CSS Grid Layout module, controlled by the `grid-auto-flow` property.

Have a go at figuring out the final position of different items in a few other layouts of your own to get a better understanding of the algorithm.

Chapter 5

How I Built a Pure CSS Crossword Puzzle

by Adrian Roworth

Recently I created a pure CSS crossword puzzle implemented using CSS grid that does not need JavaScript in order to work. It gained heavy interest pretty quickly on CodePen. As of this writing, it has more than 350 hearts and 24,000+ page views!

The great CSS Grid Garden tutorial inspired me to build something with Grid Layout features. I wondered if these features could be put to good use in building a crossword puzzle — then I thought, let's try to create the whole thing without using JavaScript.

Building the Board/Grid

So, first thing's first, let's create the board itself!

I ended up with the following basic structure, with HTML comments included to show what the different sections will accomplish:

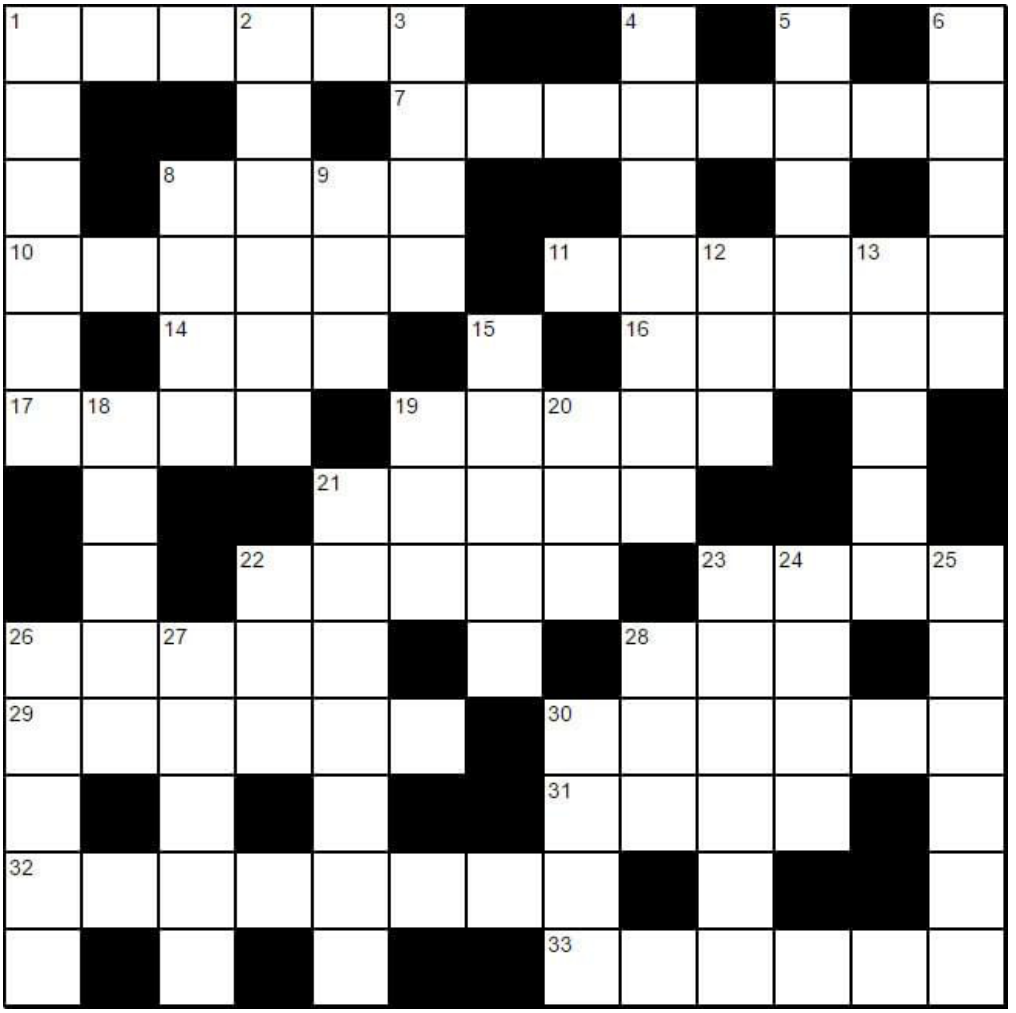
```
<div class="crossword-board-container">

  <div class="crossword-board">

 <!-- input elements go here. Uses CSS Grid as its layout
↪ -->

 <div class="crossword-board crossword-board--highlight
↪ crossword-board--highlight--across">
 <!-- highlights for valid 'across' answers go here. Uses
↪ CSS Grid as its layout -->
    </div>

 <div class="crossword-board crossword-board--highlight
↪ crossword-board--highlight-down">
 <!-- highlights for valid 'down' answers go here. Uses
↪ CSS Grid as its layout -->
    </div>

    <div class="crossword-board crossword-board--labels">
 <!-- row and column number labels go here. Uses CSS Grid
↪ as its layout -->
    </div>

    <div class="crossword-clues">

 <dl class="crossword-clues__list
↪ crossword-clues__list--across">
        <!-- clues for all the 'across' words go here -->
      </dl>

 <dl class="crossword-clues__list
```

```
↪ crossword-clues__list--down">
        <!-- clues for all the 'down' words go here -->
      </dl>

    </div>

  </div>

</div>
```

That puts our basic skeleton in place so we can add more elements and start styling things.

Using Form Elements for the Squares

The crossword puzzle I'm creating is a 13x13 grid with 44 blank spaces so I need to create 125 `input` elements each with its own ID in the format `item{row number}-{column number}`, i.e. `item4-12`. Here's what the grid will look like:



Each of the inputs will get have a `minlength` and `maxlength` of "1" to emulate the behaviour of a crossword puzzle (i.e. one letter per square). Each input will also have the `required` attribute so that HTML5 form validation will be used. I take advantage of all of these HTML5 attributes using CSS.

Using the General Sibling Selector

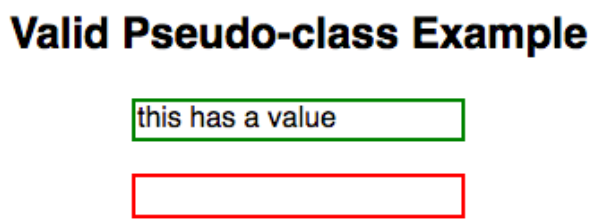
The input elements are visually laid out in groups (exactly how a crossword puzzle is). Each group of input elements represents a word in the crossword. If each of the elements in that group is valid (which can be verified using the

`:valid` pseudo selector), then we can use CSS to style an element that appears later in the DOM (using an advanced CSS selector called the general sibling selector) that will indicate that the word is correct.

Due to how sibling selectors work, and how CSS works in general, this element has to appear later in the DOM. CSS can only style elements that are after the currently selected element. It cannot look backwards in the DOM (or up the DOM tree) and style something before the current element (at the moment at least anyway).

This means I can use the `:valid` pseudo-class to style valid elements:

```
.input:valid {
  border: 2px solid green;
}
.input:invalid {
  border: 2px solid red;
}
```

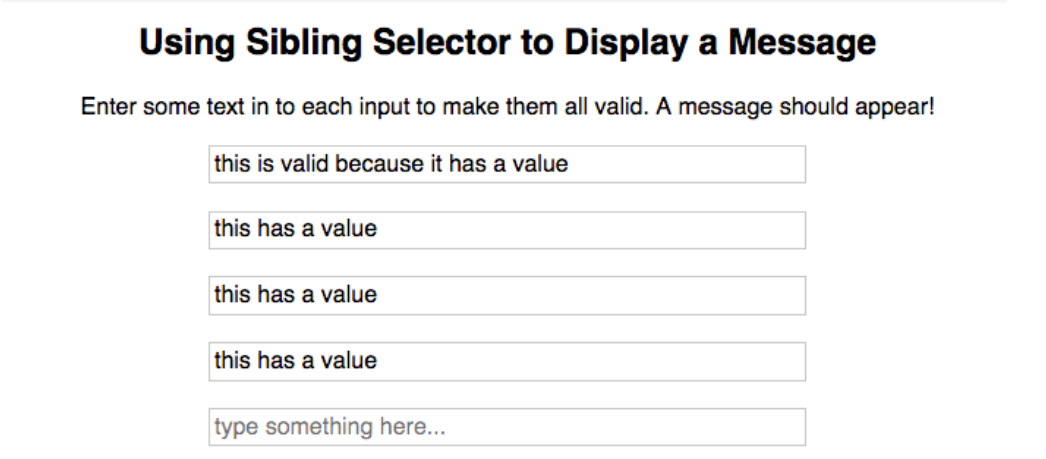


See the demo Valid Pseudo Selector Example.

To style an element later on in the DOM that is a sibling of another element, I can use the ~ (tilde/general sibling) selector, e.g. `A ~ B`. This selector will select all elements that match B, that are a sibling of A and appear after A in the DOM. For example:

```
 #input1:valid ~ #input2:valid ~ #input3:valid ~
↪ #input4:valid ~ #input5:valid ~ .valid-message {
  display: block;
}
```

With this code, if all these input elements are valid, the `valid-message` element will be displayed.



See the demo Using Sibling Selector to Display a Message.

The general sibling selector is extremely useful here. To make the crossword work, I needed to make sure that everything was laid out in a way that allowed me to take advantage of the general sibling selector.

The finished crossword example is using the above technique, starting at line 285. I've separated it out in the code block below:

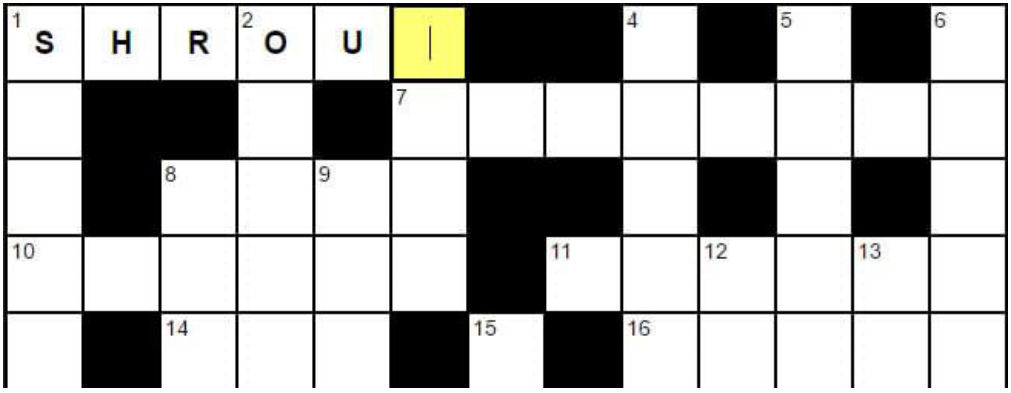
```
#item1-1:valid ~ #item1-2:valid ~ #item1-3:valid ~
#item1-4:valid ~ #item1-5:valid ~ #item1-6:valid ~
 .crossword-board--highlight
↪ .crossword-board__item-highlight--across-1 {
  opacity: 1;
```

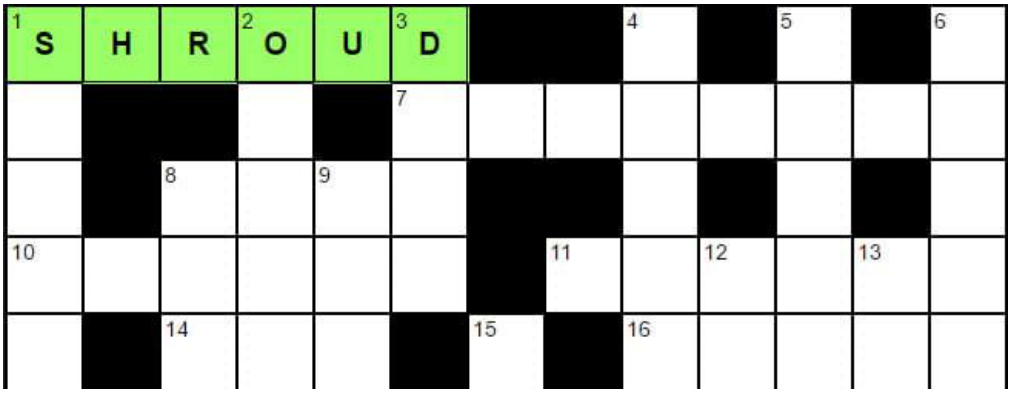
```
}
```

This part of the CSS ensures that if all these input elements are valid, then the opacity of the `.crossword-board__item-highlight--across-1` element will be changed. `.crossword-board--highlight` is a sibling of all the input elements, and `.crossword-board__item-highlight--across-1` is a child of `.crossword-board--highlight` so it's selectable with CSS!

Indicating Correct Answers

Each crossword answer (i.e. group of input elements) has a corresponding "correct answer indicator" (`.crossword-board__item-highlight--across-{{clue number}}`) grid item. These grid items are placed behind the input elements on the z-axis, and are hidden using `opacity: 0`. When a correct word is entered, then the correct answer indicator grid item is displayed by changing the `opacity` to `1`, as the pseudo-class selector snippet above demonstrates.





This technique is repeated for each "word" group of input elements. So this means manually creating each CSS rule for each of the input elements in the word group and then selecting the corresponding correct answer indicator grid item. As you can imagine, this makes the CSS get big fast!

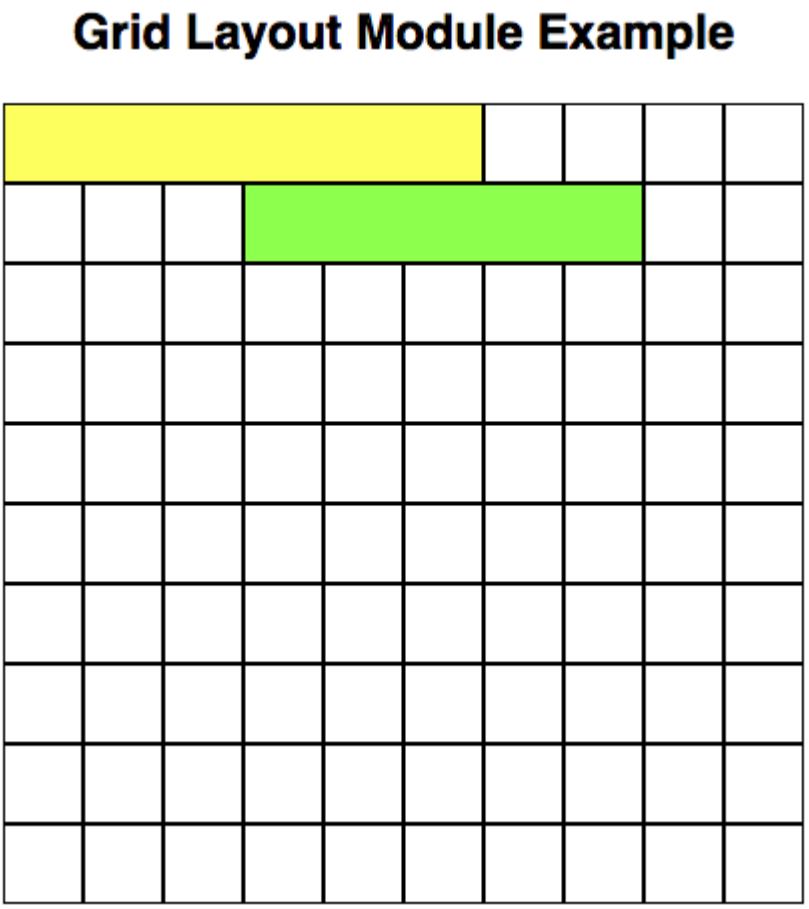
So the logical approach would be to create all the CSS rules that show/hide the correct answer indicator grid items for all the horizontal (across) clue answers. Then you would do the same for the vertical clue answers.

Challenges of the Grid System

If, like me, you are trying to use as little CSS as possible, you will quickly realise that you cannot have overlapping grid areas within the same grid system without having to explicitely declare it. They can only sit next to each other (1 across, and 1 down share a square at the top right of the board and this is not possible when using one CSS grid to layout all the correct answer indicator items).

The solution is to wrap each horizontal (across) correct answer indicator grid item in its own grid system, and each vertical (down) correct answer indicator grid item in another. This way I can still use CSS to select them (using the general sibling selector), and they will not interfere with each other and ruin the layout of the grids.

CSS Grid Layout items act similarly to inline-block elements. This basically means that if you specify two grid items to occupy the same space, then the second item will flow around the first item and appear after it in the grid.



See the demo Grid Layout Module Example.

In the above example, the first grid item is seven columns wide and spans from the first column to the seventh column. The second grid item is meant to start at the 4th column and span to the 9th column. CSS grid doesn't like this so it wraps it to the next row. Even if you specify `grid-row: 1/1` in the second item, that will take priority and then move the first grid item to the second row.

As explained above, I avoided this by having multiple grids for horizontal and vertical items. This situation can be avoided by specifying the row and column span for each element, but I used the above method to reduce the amount of CSS, and also to have a more maintainable HTML structure.

Checking for Valid Letter Input

Each input element has a `pattern` attribute with a regular expression as its value. The regular expression matches an uppercase or lowercase letter for that square:

```
<input id="item1-1" class="crossword-board__item"
       type="text" minlength="1" maxlength="1"
       pattern="^[sS]{1}$" required value="">
```

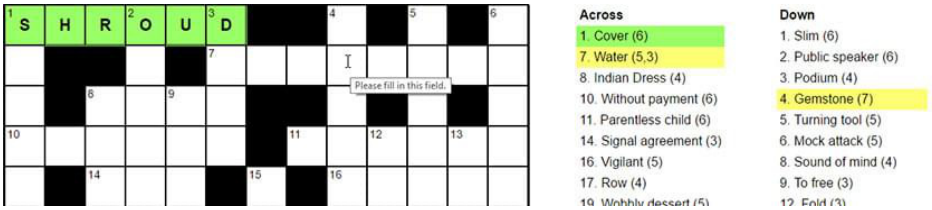
This was not ideal because the answers are in the HTML. I wanted to hide the answers in the CSS, but I could not find a way of doing this successfully. I attempted the following technique:

```
.input#item1-1[value="s"],
.input#item1-1[value="S"] {
  /* do something... */
}
```

But this won't work. The attribute selector will select the element based on what is actually inside the HTML, and doesn't consider live changes. So I had to resort to the `:valid` pseudo-class approach detailed above, and consequently (and annoyingly) exposing the answers in the HTML itself.

Highlighting the Clues on Hover

All horizontal (across) clues are wrapped in a `div`, as are the vertical (down) clues. These wrapping `div` elements are siblings of the `input` elements in the crossword grid. This is demonstrated in the HTML structure listed above in a previous code block. This makes it easy to select the correct clue(s) depending on which input element is being focused/hovered.



For this, each `input` element needs `:active`, `:focus`, and `:hover` styles to highlight the relevant clue by applying a background color when the user interacts with an `input` element.

```
 #item1-1:active ~ .crossword-clues
↪ .crossword-clues__list-item--across-1,
 #item1-1:focus ~ .crossword-clues
↪ .crossword-clues__list-item--across-1,
 #item1-1:hover ~ .crossword-clues
↪ .crossword-clues__list-item--across-1 {
  background: #ffff74;
}
```

Numbering the Clues

The numbers for the clues are positioned using a CSS Grid pattern. Here's an example of the HTML, abbreviated:

```
<div class="crossword-board crossword-board--labels">
 <span id="label-1" class="crossword-board__item-label
↪ crossword-board__item-label--1">
 <span
↪ class="crossword-board__item-label-text">1</span></span>
 <span id="label-2" class="crossword-board__item-label
↪ crossword-board__item-label--2">
 <span
↪ class="crossword-board__item-label-text">2</span></span>

  <!-- rest of the items here..... -->
```

```
</div>
```

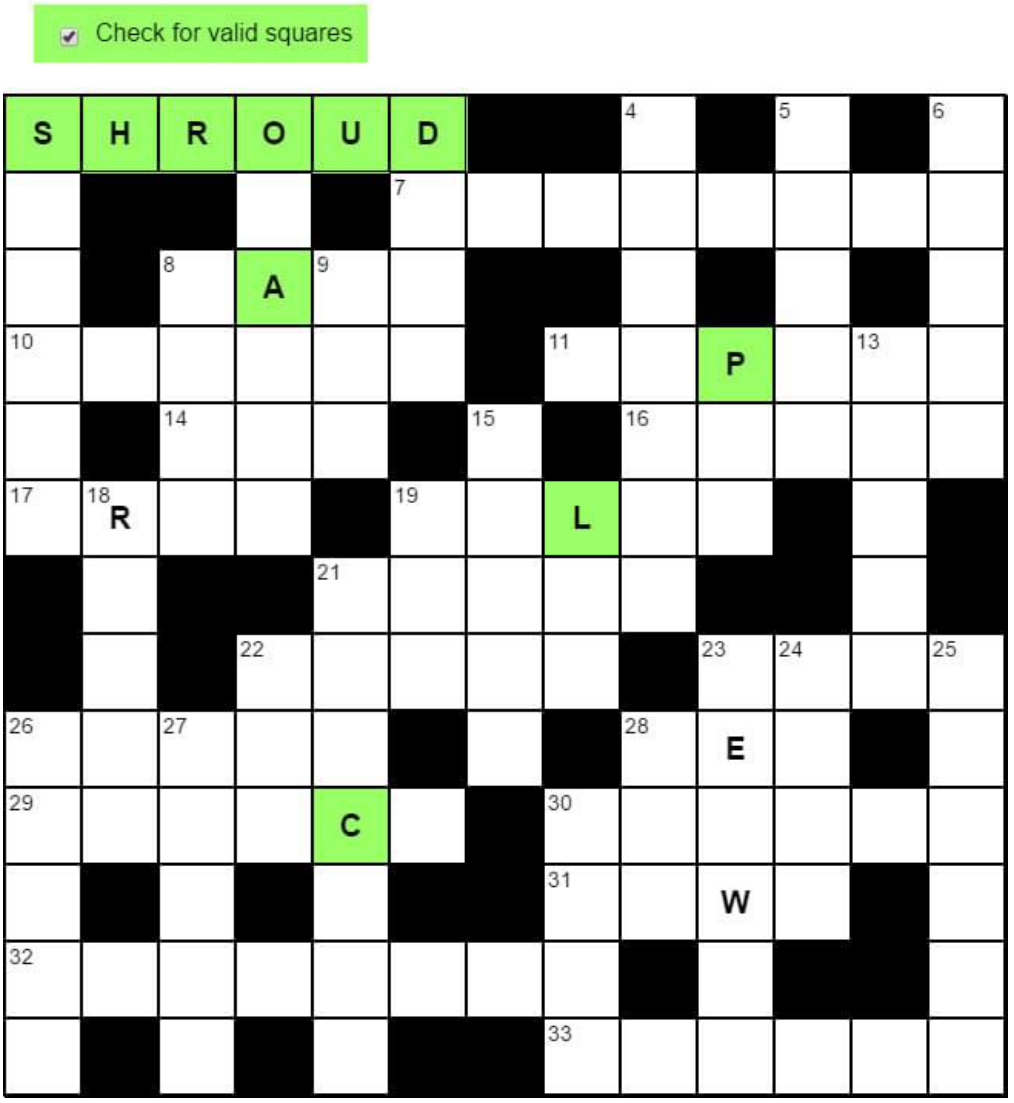
Then the CSS looks something like this:

```
.crossword-board__item-label--1 {
  grid-column: 1/1;
}

.crossword-board__item-label--2 {
  grid-column: 4/4;
}

/* etc... more items here... */
```

Each number is placed at the start position of its related group of input elements (or word). The number is then made to be the width and height of 1 grid square so that it takes up as little space as possible within the grid. It could take up even less room by implementing CSS Grid differently here, but I opted to do it this way.

The "Check for Valid Squares" Checkbox

At the top of the crossword, you'll notice a checkbox labelled "Check for valid squares". This allows the user to check if certain letters are correct, even if a given word is wrong.



Creating this is rather beautiful as it's one CSS rule that makes all the valid squares get highlighted. It's using the checkbox hack to select all valid input elements that are after the checked checkbox in the DOM.

Here is the code:

```
 #checkvaliditems:checked ~ .crossword-board-container
↪ .crossword-board__item:valid {
```

```
  background: #9aff67;
}
```

Conclusion

That covers all the main techniques used in the demo. As an exercise, this shows you how far CSS has come in recent years. There are plenty of features we can get creative with. I for one can't wait to try and push other new features to the limits!

If you want to mess around with the CSS from this article, I've put all the code examples into a CodePen collection. The full working CSS crossword Puzzle can be found here.