



CSS Grid Lanes

Native Masonry Layout for the Modern Web

SANDEEP KUMAR PATEL

CSS Grid Lanes

Native Masonry Layout for the Modern Web

Sandeep Kumar Patel

CSS Grid Lanes: Native Masonry Layout for the Modern Web

Copyright © 2026 Sandeep Kumar Patel. All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the author was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals. "CSS Grid Lanes" and `display: grid-lanes` refer to the W3C CSS Grid Layout Module Level 3 specification and its browser implementations; this book is an independent work and is not affiliated with or endorsed by the W3C, WebKit, or any browser vendor.

The information in this book is distributed on an "as is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor the publisher shall have any liability to any person or entity with respect to any loss or damage caused, or alleged to be caused, directly or indirectly, by the information contained in it.

The code that accompanies this book is released under the MIT License.

Published by Sandeep Kumar Patel.

Contents

Preface	4
Chapter 1: The Masonry Problem	9
Chapter 2: A Grid Refresher	25
Chapter 3: Your First Lanes Layout	37

Preface

For as long as the web has had photo galleries, developers have been asked to build the same layout: cards of different heights, packed into columns with no wasted space. For almost as long, we have built it with workarounds. In March 2026, Safari 26.4 shipped a way to build it in CSS itself, and this book teaches it.

CSS Grid Lanes extends the CSS Grid you already know with one new display value, `display: grid-lanes`, and one new property. It is small enough to learn quickly. It is new enough that almost nothing has been written about using it well: how the placement rule behaves with real content, what it does to keyboard and screen-reader users, and what to do in the browsers that do not support it yet. This book is about all of that, taught by building one project from a broken starting point to something you could ship.

Every claim in this book that can be checked has been checked, by a script, in a real browser engine. The listings are run, not just printed. The screenshots are captured by script from the actual project. Where the browsers disagree, or where a draft specification is still moving, the book says so and shows the measurement. You will meet a few places where the first thing we wrote turned out to be wrong when we ran it, and we left those in, because they are the best way to learn what the feature really does.

Who this book is for

This book is for front-end developers who are comfortable with HTML and CSS and have used Flexbox and CSS Grid: you know what `repeat()`, `fr` and `minmax()` do, even if you look up the details. If you have ever built a card layout and wished the columns would just pack, you are the reader.

You do not need to have used a masonry library, and you do not need to know anything about Grid Lanes. If your Grid is rusty, *Chapter 2, A Grid Refresher*, covers exactly what the rest of the book relies on, and no more. You do not need to write JavaScript to follow

the book, though two chapters include short scripts, for a fallback and for testing, and the code is explained line by line.

What this book covers

Chapter 1, The Masonry Problem, explains what masonry layout is, why CSS Grid cannot build it, what the web did instead, and where Grid Lanes stands today. It introduces Lanes Gallery, the running project.

Chapter 2, A Grid Refresher, recaps the parts of CSS Grid you need: tracks and lines, `fr`, `repeat()`, `minmax()`, `auto-fill`, placing items, and subgrid.

Chapter 3, Your First Lanes Layout, builds a grid lane from nothing, one declaration at a time, and introduces the grid axis and the stacking axis, the central idea of the book.

Chapter 4, How Items Get Packed, states the placement rule precisely, tests it against the browsers, and uses it to understand spanning, pinning and the `order` property.

Chapter 5, Taming the Flow, covers `flow-tolerance`, the one new property, and how it settles ties between lanes, including the property's changing name.

Chapter 6, Choosing an Axis: Rows vs. Columns, builds the row form of the layout, shows that everything carries over, and switches between the two with a media query.

Chapter 7, Sharing Lanes with Subgrid, uses subgrid where it works, in the grid axis, and shows why it cannot align content across lanes.

Chapter 8, Building Responsive Masonry, counts lanes, fixes a layout that overflows when narrow, and uses container queries to decide what changes at which size.

Chapter 9, The Reading Order Problem, examines what masonry does to keyboard focus and screen-reader order, measures it, and tests for it with a script.

Chapter 10, Falling Back Gracefully, detects support with `@supports`, compares the fallbacks, and builds one that works in every browser, including a small script that restores most of the masonry.

Chapter 11, Gallery Patterns in Practice, applies everything to complete patterns: a photo wall that does not jump, a mixed feed of text and images, full-width headings, and the finished gallery.

Chapter 12, Shipping the Gallery, measures performance, tests across browsers, and maps what the specification may still change.

To get the most out of this book

Grid Lanes is a young feature, so the most important requirement is a browser that supports it. At the time of writing, that means Safari 26.4 or later. Chromium-based browsers can run the listings behind an experimental flag, and *Chapter 10* explains the state of support in detail. Please check it before you rely on anything here in production.

You will also need a text editor. The project has no build step.

Software or hardware covered in the book	Requirements
Safari	Version 26.4 or later (macOS or iOS)
A second browser, to see the fallback	Any current Chrome, Edge or Firefox
Node.js and Playwright (<i>Chapters 9 to 12</i> , optional)	Node.js 20 or later; <code>npm install playwright</code>

Type the code rather than pasting it, and keep `layout.css` open in an editor next to a browser window as you read: most chapters change that one file. In Safari, the Web Inspector's grid overlay can draw the order in which lane items were placed, which makes the placement rule visible.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the download link in the next section.

Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from

<https://drive.google.com/file/d/1oHplLw378dM3GEmhqp8XO24uMKkmxRdW/view?usp=sharing> (a zip file named `css-grid-lanes-code.zip`). Unzip it and you will find:

- `lanes-gallery/` is the starting point of the running project: `index.html`, `style.css`, and a `layout.css` that is a plain grid.
- `examples/chNN/` holds the code of Chapter NN. Each listing that is a stylesheet is in its own folder, named after the listing, as a `layout.css` you can copy over the project's. The project's file for the end of a chapter is in `examples/chNN/gallery/layout.css`.
- `examples/ch11/gallery/` is the finished Lanes Gallery.
- `scripts/verify/lib.mjs` holds the placement model that Chapter 4 describes, as a short JavaScript module you can read and run.

The listings in the text are extracted from the chapters by script, so the code you download is the code you read. The code is released under the MIT License: you may use, copy and modify it in your own projects, including commercial ones, as long as you keep the copyright and license notice.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, property names, values, file names, and folder names. For example: "The `display: grid-lanes` declaration makes the container a grid lane."

A block of code is set as follows:

```
.gallery {  
  display: grid-lanes;  
  grid-template-columns: repeat(auto-fill, minmax(180px, 1fr));  
  gap: 16px;  
}
```

Bold: Indicates a new term or an important word. For example: "A **grid lane** has two axes."

TIP

Tips and important notes appear in boxes like this.

Listings, figures, and tables are numbered by chapter, such as Listing 3.1 or Figure 3.2. A *Note* is worth knowing, a *Tip* is a better way, a *Warning* is something that will bite you, and a *Sidebar* is a longer aside.

This book uses the word **lane** for each column or row that items flow into, as WebKit and MDN do. The specification's own text calls the same thing a **track**.

Screenshots are real: a script captures them from the actual running project using WebKit, the rendering engine in Safari, at a fixed size, unless a caption says otherwise.

Chapter 1: The Masonry Problem

Open almost any photo portfolio, recipe feed, or design-inspiration site and you will meet the same layout: cards of equal width but very different heights, packed into columns without wasted space. Tall images sit beside short ones. Nothing is cropped to fit a uniform rectangle. When more cards load at the bottom, everything above them stays exactly where it was.

Developers call this a **masonry layout**, after the way a mason fits stones of different sizes into a wall. For most of the web's history, building one in CSS meant giving up something you wanted, or reaching for JavaScript. On March 24, 2026, that changed: Safari 26.4 shipped **CSS Grid Lanes**, a way to write masonry as a single CSS display value, `display: grid-lanes`.

This chapter sets the scene. You will see exactly why Grid cannot do this job, what the web did instead and what each workaround costs, how Grid Lanes works at a high level, and how much of the web can use it today. Along the way you will meet Lanes Gallery, the project you will build across the rest of the book. The detailed syntax starts in *Chapter 3*. This chapter is about understanding the problem well enough that the solution feels obvious.

In this chapter, we will cover the following topics:

- What masonry layout is, and why designers keep asking for it
- Why CSS Grid cannot build it
- How the web worked around the gap, and what each workaround costs
- Enter Grid Lanes
- Where things stand today
- Meet the Lanes Gallery

Technical requirements

You need three things to follow along:

- **A browser.** Every listing that uses `display: grid-lanes`, starting with Listing 1.3, needs Safari 26.4 or later. The listings that show workarounds run in any current browser.
- **A text editor.** Lanes Gallery is plain HTML and CSS with no build step, so nothing else is required.
- **The companion code.** The `lanes-gallery/` folder holds the starting project, and the `examples/ch01/` folder holds the `layout.css` for every listing in this chapter. §1.6 explains how they fit together.

The code for this chapter is in the `examples/ch01` folder of the book's code download (see the Preface for the link).

1.1 What masonry layout is

In a masonry layout, items are placed one after another into columns, and each new item goes wherever it fits highest on the page: in the column that is currently shortest. The columns therefore end up at roughly the same height, whatever heights the individual items happen to have. You will also hear it called a **waterfall** layout, for the way content seems to pour down the page, and a **Pinterest layout**, after the site most associated with it.

The pattern is popular for three reasons, which the WebKit team laid out in 2024 when it asked the web community for feedback on the idea:

- It handles content of different aspect ratios without cropping or truncating anything to force uniform rectangles.
- It spreads content across the whole width of the page, rather than filling one column from top to bottom before starting the next.
- It lets a site load more content at the bottom without moving what is already on screen.

That makes it a natural fit for photo galleries, portfolios, product listings whose descriptions vary in length, and any feed of cards that grows as you scroll.

Masonry usually flows down columns, as in the pages above. Turn the idea on its side and you get rows of items of different widths, packed from left to right like a brick wall. This book concentrates on the column form and returns to the row form in Chapter 6.

1.2 Why CSS Grid cannot do it

You will build Lanes Gallery properly in §1.6, but you can meet it now. It is a page of nine cards, and the cards have different heights, standing in for photos of different aspect ratios. Listing 1.1 is its complete layout: a CSS Grid with as many columns as will fit, each at least 180 pixels wide, and a 16-pixel gap.

Listing 1.1. The Lanes Gallery layout as a CSS Grid

```
.gallery {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(180px, 1fr));
  gap: 16px;
}
```

Figure 1.1 shows the result.

Nothing is wrong with this Grid. It is doing exactly what Grid does: arranging items into rows and columns, where every item in a row shares that row's height. The first row is 320 pixels tall because Card 3 is. Card 1, at 260 pixels, leaves 60 pixels of dead space beneath it, and Card 2, at 160 pixels, leaves 160. Every row repeats the pattern: Card 6 leaves 80 pixels beneath it and Card 9 leaves 110. The same nine cards fill 852 pixels of height.

There is a deeper reason, and Chrome's engineers have put it well: in Grid, all items are placed before the tracks are sized. The grid decides which cell each item occupies, and only then works out how tall each row needs to be. Masonry needs the opposite order. To decide where the *next* card goes, you must already know how tall the cards placed so far are. Placement depends on size, and size is what Grid settles last.

A well-known trick approximates masonry with Grid by making the rows very short and letting each item span as many of them as it needs. It works only if you can tell the



Figure 1.1. The Lanes Gallery as a plain CSS Grid (Listing 1.1). Each row is as tall as its tallest card, so shorter cards leave gaps beneath them.

browser how tall every item is in advance. The CSS Working Group's own explainer for masonry is blunt about the limit: without knowing how tall each item will be, this layout is not possible without script. Unknown heights, such as text of any length or images from an API, are the normal case for the content masonry is used for.

1.3 How the web worked around it

Developers needed this layout long before CSS could express it, so three families of workaround grew up. Each works, and each has a price.

JavaScript libraries

The most common answer is a script that measures your cards and positions them itself. Masonry, by David DeSandro, is the best known. It places each element in the best available position based on the vertical space left, the way a mason fits stones into a wall, and it produces the same shortest-column packing that Grid Lanes does natively.

The costs are all consequences of doing layout in script:

- **Nothing is laid out until the script runs.** Until then the cards sit wherever ordinary layout puts them, so a page can visibly jump when the script finishes.
- **Heights must exist before positions can.** Masonry's documentation warns that images that have not loaded yet can throw the layout off and make items overlap, and it recommends a second library, `imagesLoaded`, to re-run the layout as each image arrives.
- **The server cannot help.** Unless you know every item's height in advance, a server has no way to send the page with the cards already in place, because the heights only exist once a browser has laid the content out.
- **Script sits outside the browser's layout engine.** The CSS Working Group's explainer notes that script-based solutions can take part in the surrounding page layout only in limited, less efficient ways than support built into the engine.

CSS multi-column layout

Multi-column layout needs no script, and it comes closest to masonry in pure CSS. Listing 1.2 asks the browser to flow the cards into three columns. The `inline-flex` declaration makes each card an inline-level box, which the browser will not split across two columns, and the margin supplies the vertical gap that `column-gap` does not.

Listing 1.2. The Lanes Gallery with CSS multi-column layout

```
.gallery {  
  column-count: 3;  
  column-gap: 16px;  
}  
  
.card {  
  display: inline-flex;  
  width: 100%;  
  margin-bottom: 16px;  
}
```

Figure 1.2 shows what it produces.

The dead space is gone, but look at the numbers. Cards 1, 2 and 3 run down the first column, 4, 5 and 6 down the second, and 7, 8 and 9 down the third. Multi-column layout fills each column from top to bottom before starting the next, so Card 2 sits *under* Card 1 rather than beside it. That is fine when the order of the cards means nothing. It is a problem when it does: a gallery sorted by newest first, or best first, reads down the left edge instead of across the top. Keyboard focus follows the same order, so tabbing through the cards means traveling to the bottom of the first column and then jumping back to the top of the second. The WebKit team cites exactly this as the difference native masonry makes.

Flexbox and Grid tricks

Flexbox can wrap a column-direction container into several columns, but only when the container has a fixed height for the items to overflow, so you have to know, or guess, how tall the content will be. It fills each column from top to bottom too, so it inherits the ordering problem above.

The trade-offs at a glance

Table 1.1 puts the four workarounds and Grid Lanes side by side.



Figure 1.2. The same cards in CSS multi-column layout (Listing 1.2). Nothing is wasted, but the numbers run down each column rather than across.

1.4 Enter Grid Lanes

Grid Lanes is the CSS Working Group's answer. It is defined in CSS Grid Layout Module Level 3, and it extends Grid rather than replacing it. Here is the idea in one sentence, close to how the specification itself puts it: a grid lane container lays items out into pre-defined tracks in one axis, as Grid does, but flows them freely in the other, as Flexbox does. For a column layout, the columns are the tracks, and the items stack down each one.

Listing 1.3 is Lanes Gallery as a grid lane. Compare it with Listing 1.1.

Table 1.1. Ways to build a masonry layout, compared

Approach	Needs JavaScript	Order of cards	Copes with unknown heights	Main cost
JavaScript library	Yes	Across, into the shortest column	Yes, once images have loaded	Layout runs after the page loads; a second library for images; nothing to render on the server
CSS multi-column	No	Down each column	Yes	Reading and tab order run down the columns
Flexbox column-wrap	No	Down each column	No	Needs a fixed container height
Grid with tiny rows	Yes, to measure	Across	Only if script measures	Heights must be known up front
Grid Lanes	No	Across, into the shortest lane	Yes	Newly shipped; limited browser support (\$1.5)

Listing 1.3. The Lanes Gallery layout as a grid lane

```
.gallery {
  display: grid-lanes;
  grid-template-columns: repeat(auto-fill, minmax(180px, 1fr));
  gap: 16px;
}
```

One word changed: `grid` became `grid-lanes`. The columns are defined exactly as before, with the same `repeat()`, `auto-fill`, `minmax()` and `gap` you already know. Figure 1.3 shows the result.



Figure 1.3. The same cards as a grid lane (Listing 1.3), captured with WebKit. Cards 1, 5 and 9 share the first lane, Cards 2, 4 and 7 the second, and Cards 3, 6 and 8 the third.

The rule that produces this is simple. Taking the cards in source order, the browser puts each one into the lane where it can sit highest, which means the lane that is currently shortest. Cards 1, 2 and 3 fill the three lanes side by side. Card 2 is the shortest so far, so Card 4 goes beneath it. That makes the second lane taller than the first, so Card 5 goes beneath Card 1. Card 6 lands beneath Card 3, Card 7 beneath Card 4, Card 8 beneath Card 6, and Card 9 beneath Card 5. The numbers run across the page instead of down it,

and the three lanes end within 30 pixels of one another. The same nine cards now fit in 692 pixels of height instead of 852.

WebKit's announcement gives the idea its name: picture the cards as cars in traffic, each changing lanes to end up in the lane that gets it furthest ahead.

NOTE

Lanes and tracks. WebKit, MDN and this book say **lane** for each column (or row) that items flow into. The specification's own text says **track**, because these are the same tracks Grid defines. They mean the same thing.

There is a second benefit that matters for infinite feeds. Because each card is placed the moment it arrives, appending a tenth card to Lanes Gallery leaves the first nine exactly where they were.

Everything you know about *defining* tracks carries over. What is new is a single property, `flow-tolerance`, which controls how the browser breaks ties between lanes of nearly equal height (Chapter 5). Define rows instead of columns and you get the brick-wall form (Chapter 6). Chapter 4 covers spanning items across lanes and pinning them to specific ones.

1.5 Where things stand today

Grid Lanes is young, and whether you can use it depends on your audience. Table 1.2 shows the road here, in brief.

As of September 2026, support breaks down like this:

- **Safari 26.4 and later**, on macOS and iOS, supports `display: grid-lanes` by default. According to caniuse.com, about one in ten users worldwide were on a browser with default support.
- **Chrome and Edge** support it behind a flag. Tested directly in a Chromium 153 build: turning on the experimental web platform features flag enables the final `display: grid-lanes` syntax and lays out Lanes Gallery identically to Safari.

Table 1.2. The road to Grid Lanes

When	What happened
January 2020	Mozilla proposes masonry for CSS Grid, and Firefox Nightly gets an experimental implementation behind a flag
2022	WebKit begins its own implementation
January 2025	The CSS Working Group resolves to build masonry from Grid's existing template and placement properties
July 2025	Chrome and Edge land a prototype behind a flag, available from version 140
December 2025	WebKit announces the name grid-lanes and ships it in Safari Technology Preview
January 2026	<code>item-tolerance</code> is renamed <code>flow-tolerance</code> (January 7), and the W3C publishes a Working Draft (January 21)
March 24, 2026	Safari 26.4 ships Grid Lanes

- **Firefox** has an older experiment behind a flag that predates the rename. It understands `grid-template-rows: masonry`, not `display: grid-lanes`, so Listing 1.3 does nothing there even with the flag on.

What does an unsupported browser do with Listing 1.3? It does not understand the `display` value, so it discards the declaration. The container stays an ordinary block, `grid-template-columns` and `gap` have nothing to act on, and the cards stack in a single full-width column (Figure 1.4).

Chapter 10 shows how to detect support and fall back gracefully, so that visitors on other browsers still get a good page.



Figure 1.4. Listing 1.3 in Chromium without the experimental flag (first three cards). The unrecognized display value is dropped, and the cards stack one per row with no gaps.

WARNING

Grid Lanes is a Working Draft, not a finished standard. As of the September 2, 2026 editor's draft of the specification, even how a lanes container's orientation is controlled is still being decided. This book sticks to the stable core (the display value, tracks, gaps, spanning, and `flow-tolerance`) and flags anything still in flux where it comes up. One name has already changed twice: the property this book calls `flow-tolerance` began as `item-tolerance`, and the October 2026 editor's draft of the specification calls it `fit-tolerance`. Safari accepts both new names and Chromium's flagged build accepts only `flow-tolerance`, so the book uses `flow-tolerance` (*Chapter 5* has the details). Treat the syntax here as current for the time of writing, and check the specification before you ship.

Here is how this book handles a feature that is still settling:

- Screenshots are captured by script with WebKit, the rendering engine in Safari, at a fixed size, unless a caption says otherwise. Figure 1.4 is the deliberate exception.
- Every listing was run in a real engine before it was printed, and the project's code is extracted from the chapters by script, so it cannot drift from the text.
- Chapter 9 covers the accessibility problems masonry brings, and Chapter 10 covers detection and fallbacks.

1.6 Meet the Lanes Gallery

Lanes Gallery is a photo and portfolio feed, and you will grow it chapter by chapter until it is ready to ship. Its files are deliberately few:

- `index.html` holds the nine cards. Each card's caption is a link; that does not matter yet, but it will in *Chapter 9*. Cards 1 and 6 also carry a `featured` class that no style uses until *Chapter 4*.
- `style.css` holds appearance only: colors, type, and the card styling.
- `layout.css` holds how the cards are arranged. **This is the one file the book changes as it goes**, and it starts life as Listing 1.1.

The `lanes-gallery/` folder always holds this starting point. After each chapter that changes the project, you will find its finished `layout.css` in `examples/chNN/gallery/`, ready to copy over `lanes-gallery/layout.css`.

Each card gets its height from a custom property (`--h`), standing in for a photo's aspect ratio, and a gradient fill instead of a picture, so that every figure in this book is reproducible. Nothing in the layout code depends on the cards being gradients, so you can substitute real photos in your own project.

Let's try it now. Open `lanes-gallery/index.html` in any browser and you will see Figure 1.1. Then, in Safari 26.4 or later, change `display: grid` to `display: grid-lanes` in `layout.css` and reload. The gaps close up, and you have seen the whole idea in one line.

TIP

Safari's Web Inspector can show you what a grid lane is doing. From Safari 26.4, its grid overlay supports Grid Lanes containers: it can draw the order in which items were placed and show the gaps between them. Safari 27 adds a Grid Lanes badge in the Elements tab too, so a lanes container is easy to spot while you are browsing the DOM. It is the quickest way to watch the placement rule from §1.4 at work.

Here is where the project goes from here:

- **Chapter 3** turns `layout.css` into a proper grid lane and explains every line.
- **Chapter 4** states the placement rule exactly, and gives some cards more room by spanning them across lanes.
- **Chapter 5** tunes `flow-tolerance` so the layout behaves with real photos.
- **Chapter 6** adds a row-based view of the gallery alongside the column-based one.
- **Chapter 7** uses subgrid where it works, to line the featured cards' captions up with the lanes, and shows where it cannot.
- **Chapter 8** makes the gallery responsive.
- **Chapter 9** audits it for the reading-order problems masonry brings, and **Chapter 10** gives it a fallback for browsers without support.

- **Chapter 11** brings the patterns from earlier chapters together into a feature-complete gallery, and **Chapter 12** ships it.

1.7 Summary

- A masonry layout packs items of different heights into columns, placing each new item in the shortest one, so nothing is cropped and there is no dead space.
- CSS Grid cannot build it, because Grid places items before it sizes its tracks, and every item in a row shares that row's height.
- The old workarounds each have a cost. JavaScript libraries lay out late and depend on images loading. Multi-column layout reads down the columns instead of across. Flexbox needs a fixed height. Grid tricks need known heights.
- Grid Lanes adds `display: grid-lanes` to CSS Grid. You define tracks as you already do, and the browser places each item in the shortest lane. Switching Lanes Gallery from Listing 1.1 to Listing 1.3 is a one-word change.
- It is a Working Draft. Safari 26.4 and later ships it, Chromium browsers have it behind a flag, Firefox has only the older syntax, and other browsers ignore it, which is why Chapter 10 matters.
- Lanes Gallery is the project you will build throughout this book, and `layout.css` is the file that changes.

Questions

1. Why can CSS Grid not build a masonry layout, however its tracks are defined?
2. Multi-column layout packs cards without dead space. What is the main problem with using it for a gallery that is sorted by newest first?
3. What changes between Listing 1.1 and Listing 1.3, and what do the other declarations do?
4. What does a browser that does not support Grid Lanes do with `display: grid-lanes` ?
5. Which file does the book change from chapter to chapter, and what does it hold?

The answers are in the *Assessments* appendix.

Further reading

- *The Field Guide to CSS Grid Lanes*, WebKit: gridlanes.webkit.org
- *Introducing CSS Grid Lanes*, WebKit blog, December 19, 2025: webkit.org/blog/17660
- *CSS Grid Layout Module Level 3*, W3C: w3.org/TR/css-grid-3
- *Grid lanes layout*, MDN Web Docs: developer.mozilla.org/en-US/docs/Web/CSS/Guides/Grid_layout/Grid_lanes

Chapter 2: A Grid Refresher

Everything in Grid Lanes is built on CSS Grid. The tracks of a grid lane are defined with the same properties as the tracks of a grid, items are placed into them with the same placement properties, and the gaps between them come from the same `gap`. If you can read a `grid-template-columns` declaration at a glance, Grid Lanes will feel like a small step. If you cannot yet, this chapter gets you there.

This is a refresher, not a course. We cover only the parts of Grid that the rest of the book leans on, and we run each of them against the Lanes Gallery so that you can see what it does. If you use Grid every day, skim the headings, read the two notes on rows and on `auto-fit`, and move on to *Chapter 3*.

In this chapter, we will cover the following topics:

- Tracks, lines, and cells
- Sizing tracks with `fr`, `repeat()`, and `minmax()`
- Letting the browser count the tracks with `auto-fill` and `auto-fit`
- Placing items with spans and line numbers
- Implicit tracks and alignment
- Subgrid

Technical requirements

Every listing in this chapter is plain CSS Grid, so any current browser will run it: Chrome, Edge, Firefox, or Safari. You need the `lanes-gallery/` folder from the book's code download. The finished `layout.css` for each listing is in `examples/ch02/`, in a folder named after the listing. To try one, copy its `layout.css` over `lanes-gallery/layout.css` and reload `index.html`.

The widths quoted in this chapter come from an 800-pixel-wide browser window. The gallery fills the window minus 24 pixels of padding on each side, which makes it 752

pixels wide. If your window is a different width, your numbers will be different too, but the ratios will be the same.

2.1 Tracks, lines, and cells

A **grid container** is any element with `display: grid`. Its children become **grid items**, and the container divides its space into **tracks**: columns running vertically and rows running horizontally. The boundaries between tracks are numbered **grid lines**, and the rectangle where a row and a column cross is a **cell**.

Two ideas matter more than the rest for this book. The first is that lines are numbered from 1, so a grid with three columns has four vertical lines, numbered 1 to 4. You can also count backward from the end: line -1 is the last line, and line -2 is the one before it. The second idea is that the tracks are shared. Every item in a column shares that column's width, and every item in a row shares that row's height.

NOTE

Rows are shared, and that is the whole masonry problem. A row is exactly as tall as the tallest item in it, and all of the items in the row must fit within that height. We met the consequence in *Chapter 1*: a short card sitting next to a tall one leaves a gap beneath it. Grid Lanes changes this one thing and nothing else, so it is worth keeping in mind as you read the rest of the chapter.

Figure 2.1 marks the lines of a three-column grid on the Lanes Gallery. The picture is captured from the real page, and the line markers are placed from the track sizes the browser reports, so they cannot disagree with the layout.

2.2 Sizing tracks

You define columns with `grid-template-columns` and rows with `grid-template-rows`. Each value is a list of track sizes, one per track. Listing 2.1 gives the Lanes Gallery three columns: a fixed 200 pixels, then two flexible ones.

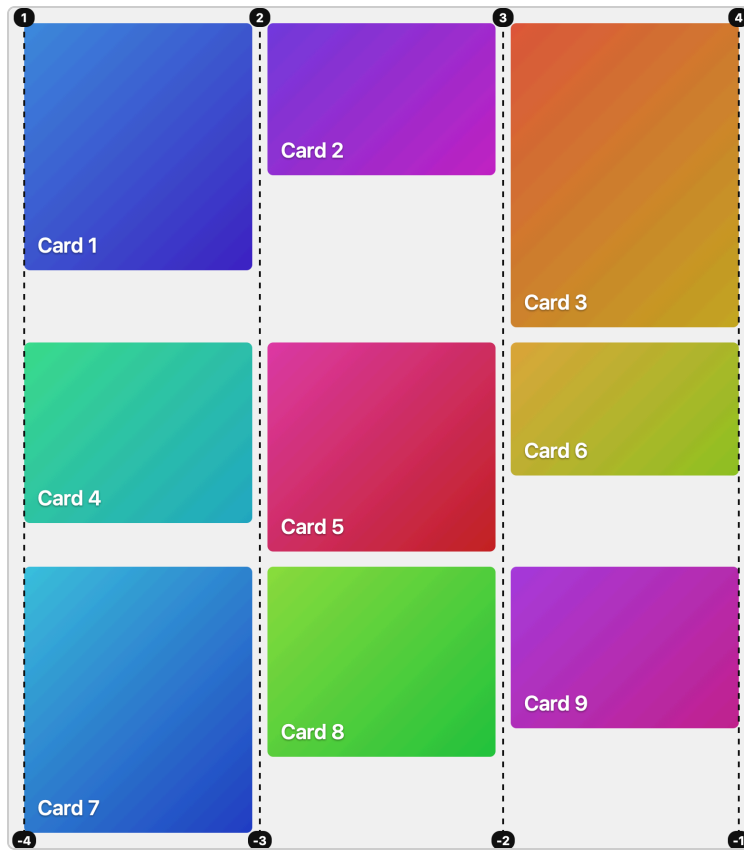


Figure 2.1. The grid lines of a three-column grid (Listing 2.2). Four vertical lines are numbered 1 to 4 from the left, and the same lines count -4 to -1 from the right.

Listing 2.1. Three columns, one fixed and two flexible

```
.gallery {
  display: grid;
  grid-template-columns: 200px 1fr 2fr;
  gap: 16px;
}
```

The `fr` unit is a fraction of the *free space*, which is whatever is left in the container once the fixed tracks and the gaps are taken out. The gallery is 752 pixels wide and the two gaps use 32 of them, so 720 pixels remain. The fixed column takes 200, which leaves 520

to divide between three fractions: one for the second column and two for the third. The second column is therefore 173.3 pixels wide and the third is 346.7.

Writing the same size many times is tedious, so `repeat()` does it for you. Listing 2.2 is three equal columns.

Listing 2.2. Three equal columns with `repeat()`

```
.gallery {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 16px;
}
```

Each of the three tracks gets a third of 720 pixels, which is 240. This is the shape you will see most often in this book: `repeat()` with a count, a size, and a gap.

The third tool is `minmax(min, max)`, which sets a floor and a ceiling for a track. `minmax(180px, 1fr)` means "never narrower than 180 pixels, and otherwise take an equal share of the free space". On its own it is not very interesting, because the number of tracks is still fixed. Combined with a way of counting tracks automatically, it becomes the most useful line of Grid.

2.3 Letting the browser count the tracks

`repeat()` also accepts the keywords `auto-fill` and `auto-fit` in place of a number. The browser then works out how many tracks of the given size fit in the container, and creates that many. Listing 2.3 is the layout the Lanes Gallery has used since *Chapter 1*.

Listing 2.3. As many columns of at least 180 pixels as will fit

```
.gallery {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(180px, 1fr));
  gap: 16px;
}
```

The count is the largest number of 180-pixel columns that fit once the gaps are included. At 752 pixels, three columns need $3 \times 180 + 2 \times 16 = 572$ pixels and four need $4 \times 180 + 3 \times 16 = 768$, which is too wide, so the browser creates three. They then grow from 180 pixels to fill the container, giving 240 pixels each. If the window grows to 1,000 pixels, the gallery stops at its 900-pixel limit and now fits four columns of 213 pixels each. *Chapter 8* builds on exactly this behavior to make the gallery responsive without a single media query.

The difference between the two keywords appears only when there are fewer items than tracks. Listing 2.4 shows only the first two cards and asks for `auto-fit`.

Listing 2.4. Collapsing empty columns with auto-fit

```
.gallery {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(180px, 1fr));
  gap: 16px;
}

/* Show only two cards, to expose the difference */
.card:nth-child(n + 3) {
  display: none;
}
```

With `auto-fill`, the browser still creates three columns and leaves the third empty, so the two cards are 240 pixels wide each. With `auto-fit`, the empty columns collapse to nothing and the two cards share the whole width, 368 pixels each. In a grid, then, `auto-fill` keeps the empty tracks and `auto-fit` removes them.

NOTE

Do not expect auto-fit to behave the same way in a grid lane. We tested the two keywords in Grid Lanes with two cards. WebKit gave identical results for both, with the cards 240 pixels wide. Chromium with its experimental flag collapsed the empty lanes, as Grid does, and made the cards 368 pixels wide. *Chapter 8* returns to this: in a grid lane, use `auto-fill`.

2.4 Placing items

By default, a grid places its items automatically: each item takes the next free cell, filling a row from left to right before starting the next. For most items that is exactly what you want. When it is not, you can tell an item where to go with `grid-column` and `grid-row`, which accept line numbers and spans.

The `span` keyword makes an item cover several tracks. Listing 2.5 makes the two featured cards cover two columns each.

Listing 2.5. Spanning two columns

```
.gallery {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(180px, 1fr));
  gap: 16px;
}

.featured {
  grid-column: span 2;
}
```

Cards 1 and 6 now measure 496 pixels wide, which is two columns and the gap between them. Figure 2.2 shows the result, and with it the cost: because Grid fills its cells in order, each featured card pushes a neighbor along, and the rows keep their shared heights, so gaps open up between the cards.

Line numbers give you exact control. `grid-column: 2 / 4` means "start at line 2 and end at line 4", which covers the second and third columns. The shorthand `grid-column: 1 / -1` runs from the first line to the last, so it makes an item as wide as the whole explicit grid, whatever the number of columns. Listing 2.6 uses it to give each featured card a row of its own.

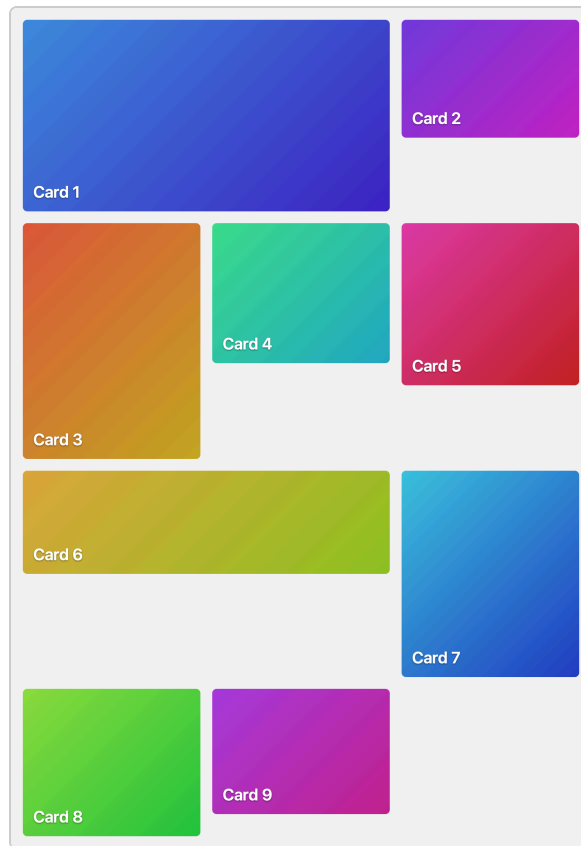


Figure 2.2. The gallery with the two featured cards spanning two columns (Listing 2.5). The rows still share one height each, so short cards sit above empty space.

Listing 2.6. Spanning every column with line numbers

```
.gallery {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 16px;
}

.featured {
  grid-column: 1 / -1;
}
```

Both featured cards are now 752 pixels wide. Note that `-1` counts from the end of the *explicit* grid, the tracks you defined. We will meet that qualification again, because it is one of the places where a grid lane behaves differently.

TIP

Prefer `span` when you mean "this item is wider than the rest" and line numbers when you mean "this item belongs in this place". An item that spans two columns keeps working when the number of columns changes. An item pinned to `2 / 4` does not.

2.5 Implicit tracks and alignment

A grid creates tracks you did not ask for whenever an item needs a cell that the template does not provide. These are **implicit tracks**. In our galleries the template defines only columns, so every row is implicit. Rows size themselves to their content unless you say otherwise with `grid-auto-rows`. Listing 2.7 fixes every row at 100 pixels and lets the cards fill them.

Listing 2.7. Implicit rows of a fixed height

```
.gallery {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(180px, 1fr));
  grid-auto-rows: 100px;
  gap: 16px;
}

.card {
  height: auto;
}
```

The `height: auto` rule is needed because the Lanes Gallery gives each card an explicit height. Without it, the card would ignore the row and keep its own height. With it, each card stretches to fill its 100-pixel row, which is Grid's default behavior, and the nine cards occupy three rows, 332 pixels in total.

That stretching is controlled by alignment. `align-items` positions items inside their cells vertically and `justify-items` does the same horizontally. The default, `stretch`, fills the cell; `start`, `center` and `end` pin the item to one side of it. In Listing 2.8, the tallest card in the first row is Card 3, at 320 pixels. Aligning to the `end` moves Card 1, which is 260 pixels tall, down by the 60 pixels it was short.

Listing 2.8. Bottom-aligning items in their cells

```
.gallery {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(180px, 1fr));
  gap: 16px;
  align-items: end;
}
```

Alignment moves the empty space around; it does not remove it. The 60 pixels are still there, now above the card instead of below. This is another way of seeing why alignment cannot solve the masonry problem.

2.6 Subgrid

Nested grids have a classic problem. Suppose each card in a row has a title, some text, and a "Read more" link, and the cards' titles and text are different lengths. Each card is its own grid, so its rows are sized independently, and the "Read more" links end up at different heights.

Subgrid solves this. A grid item that is itself a grid can declare `grid-template-rows: subgrid` (or `grid-template-columns: subgrid`) to use the parent's tracks rather than creating its own. Its children then take part in the parent's track sizing, and the cards line up. Listing 2.9 is the markup for three such cards, and Listing 2.10 is the layout.

Listing 2.9. Three cards, each with a title, text, and a link

```
<main class="gallery">
  <article class="card">
    <h2>Short</h2>
    <p>One short line.</p>
    <a href="#">Read more</a>
```

```

</article>
<article class="card">
  <h2>A title long enough to wrap onto a second line</h2>
  <p>This card has several lines of text, so it is the tallest of the three.</p>
  <a href="#">Read more</a>
</article>
<article class="card">
  <h2>Medium</h2>
  <p>A little more text than the first card, but not as much as the second.</p>
  <a href="#">Read more</a>
</article>
</main>

```

Listing 2.10. Each card borrows three rows from the grid

```

.gallery {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  column-gap: 16px;
}

.card {
  display: grid;
  grid-row: span 3;
  grid-template-rows: subgrid;
}

```

Each card spans three of the parent's rows, one for the title, one for the text, and one for the link, and declares those rows as its own with `subgrid`. All three titles now share a row, so the row is as tall as the longest title, and likewise for the text and the links. Figure 2.3 shows the links at the same height in all three cards.

Subgrid has been available in every major browser since Safari 16, Firefox 71, and Chrome and Edge 117, so you can rely on it. Hold on to this picture, because *Chapter 7* returns to it: the trick works because the cards share *rows*, and a grid lane does not have shared rows in its stacking direction.



Figure 2.3. Three cards sharing three rows through subgrid (Listings 2.9 and 2.10). The three "Read more" links sit at the same height, whatever the length of the text above them.

2.7 Summary

- A grid divides its container into tracks, numbered by grid lines. Lines count from 1 at the start and from -1 at the end.
- Tracks are shared: every item in a row has that row's height. That is the masonry problem in one sentence.
- `fr` divides the free space, `repeat()` avoids repetition, and `minmax()` sets a floor and a ceiling for a track.
- `repeat(auto-fill, minmax(180px, 1fr))` lets the browser decide how many columns fit. `auto-fit` additionally collapses the empty ones.
- `grid-column` and `grid-row` place items by span or by line number, and `1 / -1` covers the whole explicit grid.
- Rows you do not define are implicit; `grid-auto-rows` sizes them. Alignment moves space around but never removes it.
- Subgrid lets a nested grid borrow its parent's tracks, which lines up content across siblings.

Questions

1. If a 752-pixel-wide container has `grid-template-columns: 200px 1fr 2fr` and a 16-pixel gap, how wide is the third column?
2. What is the difference between `auto-fill` and `auto-fit`, and when does it show?

3. What does `grid-column: 1 / -1` do, and what does `-1` refer to?
4. Why does `align-items: end` not remove the gaps beneath short cards?
5. What does `grid-template-rows: subgrid` let a card do, and why does it work in a grid but, as *Chapter 7* will show, not across the lanes of a grid lane?

The answers are in the *Assessments* appendix.

Further reading

- *CSS Grid Layout Module Level 1*, W3C: [w3.org/TR/css-grid-1](https://www.w3.org/TR/css-grid-1)
- *Subgrid*, CSS Grid Layout Module Level 2, W3C: [w3.org/TR/css-grid-2](https://www.w3.org/TR/css-grid-2)
- *Basic concepts of grid layout*, MDN Web Docs: developer.mozilla.org/en-US/docs/Web/CSS/Guides/Grid_layout/Basic_concepts

Chapter 3: Your First Lanes Layout

In *Chapter 1*, we changed one word in `layout.css` and watched the gaps in the Lanes Gallery close. That showed that Grid Lanes works, but not why, and it left you with a one-word change you could not yet explain. This chapter explains it.

We will build a grid lane from nothing, one declaration at a time, and look at what each declaration does to the layout. Along the way we will meet the single most useful idea in the book: a grid lane has two axes, one that behaves like Grid and one that behaves like Flexbox. Once you have that picture, the rest of the specification is a matter of detail.

In this chapter, we will cover the following topics:

- One display value, two spellings
- A grid lane with no lanes
- Defining the lanes
- The grid axis and the stacking axis
- Adding gaps
- Inline grid lanes
- Project checkpoint: the Lanes Gallery

Technical requirements

Every listing in this chapter uses `display: grid-lanes`, so you need Safari 26.4 or later (see *Chapter 1* for the state of support in other browsers). You also need the `lanes-gallery/` folder from the book's code download. The finished `layout.css` for each listing is in `examples/ch03/`, in a folder named after the listing; copy it over `lanes-gallery/layout.css` and reload `index.html` to try it.

As in *Chapter 2*, the widths quoted here come from an 800-pixel-wide window, which makes the gallery 752 pixels wide.

3.1 One display value, two spellings

Grid Lanes adds two values to the `display` property. `display: grid-lanes` makes an element a block-level grid lane container, just as `display: grid` makes a block-level grid container. `display: inline-grid-lanes` makes it an inline-level one, matching `inline-grid`. Both are defined in CSS Grid Layout Module Level 3, and a browser that supports one supports the other. We use `grid-lanes` nearly everywhere in this book and look at the inline form in §3.6.

A grid lane container has the same relationship to its children as a grid container has. Its children become **grid lane items**, and the container lays them out in **lanes**. The specification calls the lanes *tracks*, because they are the same tracks that CSS Grid defines. This book says *lane* when it means the column or row that items flow into, and *track* when it means the sizing of one.

3.2 A grid lane with no lanes

What is the smallest possible grid lane? Listing 3.1 is just the display value.

Listing 3.1. A grid lane container with no lanes defined

```
.gallery {
  display: grid-lanes;
}
```

You might expect nothing to happen, since there is nothing in the layout to say how many lanes to make. Something does happen: the container makes one lane, as wide as the container itself, and stacks every card in it. The nine cards are each 752 pixels wide, and they sit directly on top of one another, so the gallery is 1,940 pixels tall, which is the sum of the nine card heights.

That looks exactly like the unsupported-browser fallback in Figure 1.4, and it is a useful thing to notice. A grid lane with no lanes is a stack. If you ever see a stack when you expected lanes, check that the lanes are actually defined.

3.3 Defining the lanes

Lanes are defined the way Grid's tracks are, with `grid-template-columns`. Listing 3.2 asks for three equal lanes.

Listing 3.2. Three equal lanes, with no gap

```
.gallery {
  display: grid-lanes;
  grid-template-columns: repeat(3, 1fr);
}
```

Now the cards spread into three lanes of 250.7 pixels each, and the layout takes on the shape you saw in *Chapter 1*. Cards 1, 2 and 3 take the three lanes, Card 4 goes beneath the shortest of them, and so on, exactly as described in §1.4. The gallery is 660 pixels tall. With no gap defined, the cards touch.

Everything you learned in *Chapter 2* about defining tracks applies unchanged: `fr`, `repeat()`, `minmax()`, and `auto-fill`. The next listing adds a gap, and Figure 3.1 shows the first three steps side by side.

Listing 3.3. Three equal lanes with a 16-pixel gap

```
.gallery {
  display: grid-lanes;
  grid-template-columns: repeat(3, 1fr);
  gap: 16px;
}
```

3.4 The grid axis and the stacking axis

Look again at the right-hand picture in Figure 3.1. The cards line up neatly side by side, but not from top to bottom: each lane has its own height. That difference is what separates a grid lane from a grid, and it comes from the fact that a grid lane has two different axes.

The **grid axis** is the axis along which you define the lanes. In Listing 3.3 you defined columns, so the grid axis is the horizontal one. In this axis, a grid lane behaves exactly

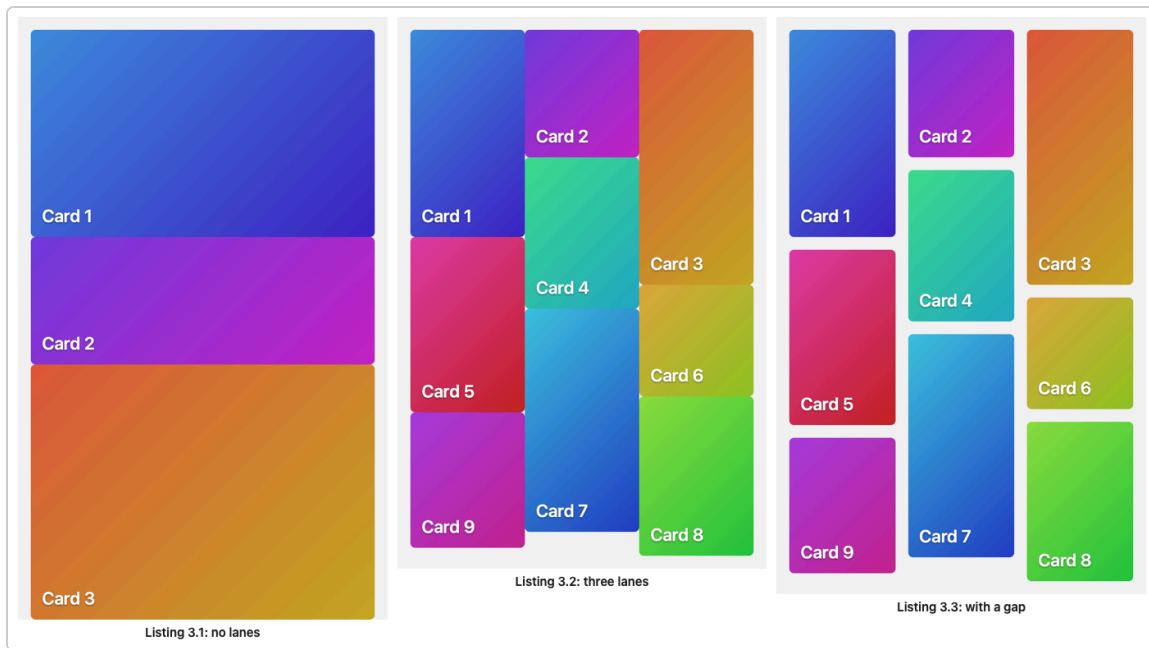


Figure 3.1. How the layout builds up. Left: Listing 3.1, one lane (first three cards shown). Middle: Listing 3.2, three lanes with no gap. Right: Listing 3.3, the same lanes with a 16-pixel gap.

like a grid: the lanes have the sizes you gave them, and each item is as wide as its lane, or as wide as the lanes it spans.

The **stacking axis** is the other one, the vertical axis in this layout. In this axis, a grid lane behaves like Flexbox. There are no rows to share, so each item is as tall as its own content wants it to be, and the items in a lane simply stack one beneath the other. Each lane is independent, and ends wherever its items end.

Figure 3.2 labels both axes on the real layout of Listing 3.3.

This split explains everything else in the chapter, so it is worth spelling out which properties belong to which axis. Table 3.1 does that for the column layout we have been building, and for its rotated twin, which *Chapter 6* covers in full.

You choose the orientation by choosing which template to write. If you define `grid-template-columns`, you get column lanes. If you define only `grid-template-rows`, you



Figure 3.2. The two axes of a grid lane (Listing 3.3). Lanes are defined along the grid axis. Items stack along the stacking axis.

get row lanes. If you define both, the columns win and the rows are ignored, which we confirmed by running a container that had both defined; avoid it, since it is a sign that you mean something else.

Table 3.1. What belongs to each axis in a column layout and in a row layout

	Column lanes (<code>grid-template-columns</code>)	Row lanes (<code>grid-template-rows</code>)
Grid axis	Horizontal	Vertical
Stacking axis	Vertical	Horizontal
An item's size in the grid axis	The width of its lane (or lanes)	The height of its lane (or lanes)
An item's size in the stacking axis	Its content height	Its content width
Gap between lanes	<code>column-gap</code>	<code>row-gap</code>
Gap between items in a lane	<code>row-gap</code>	<code>column-gap</code>

WARNING

How a grid lane's orientation is chosen is still being decided. As of the October 2026 editor's draft of the specification, the property that will control it explicitly is marked as undecided, and the Chromium build we tested, behind its flag, already has a `grid-lanes-direction` property that Safari does not. Choosing the orientation through `grid-template-columns` or `grid-template-rows` works in every engine that supports Grid Lanes, so that is the method this book uses.

3.5 Adding gaps

You will have noticed that Listing 3.3 uses the same `gap` property as every grid. In a grid lane, the gap does both of its jobs: `column-gap` separates the columns, and `row-gap` separates the rows. In a column layout, though, the vertical dimension has no rows, so what does `row-gap` do?

It separates the items within each lane. Table 3.1 gives the rule: in a column layout, `column-gap` is the gap between lanes and `row-gap` is the gap between items in a lane. Listing 3.4 gives them different sizes so that you can see which is which.

Listing 3.4. Different gaps in the two directions

```
.gallery {
  display: grid-lanes;
  grid-template-columns: repeat(3, 1fr);
  column-gap: 8px;
  row-gap: 30px;
}
```

Figure 3.3 shows the result. The lanes are 8 pixels apart, and cards within a lane are 30 pixels apart. The gallery is 720 pixels tall, 60 pixels taller than in Listing 3.2, because at least one lane contains two 30-pixel gaps.

Keep this rule in mind: the gap *between lanes* always uses the gap property that runs along the grid axis, and the gap *between items* uses the one that runs along the stacking axis. For column lanes, that means `column-gap` and `row-gap`. For row lanes, it reverses, as the right-hand column of Table 3.1 says.

TIP

Most of the time you want the same gap in both directions, so use the `gap` shorthand and forget about the distinction. Reach for the longhands only when you want a deliberately different rhythm, such as tight rows of text cards in wide lanes.

3.6 Inline grid lanes

`display: inline-grid-lanes` makes the container itself inline-level. It sits in a line of text like an inline element, and, like `inline-grid` or `inline-block`, it is only as wide as its content needs. Listing 3.5 sets three fixed lanes of 120 pixels and makes the gallery inline.



Figure 3.3. Different gaps in the two directions (Listing 3.4). The lanes are 8 pixels apart; the cards within a lane are 30 pixels apart.

Listing 3.5. An inline grid lane container

```
.gallery {
  display: inline-grid-lanes;
  grid-template-columns: repeat(3, 120px);
  gap: 16px;
}
```

The gallery is now 392 pixels wide, which is the three 120-pixel lanes plus the two gaps. A block-level container would have stretched to the full 752 pixels. Browsers that support the single-word value also accept the two-word spelling `display: inline grid-lanes` (we tested both), which is the form modern CSS prefers for inline-level display values.

You will rarely need either: an inline container is for cases such as a small masonry widget placed in the flow of a paragraph. We include it mainly so that you recognize it.

NOTE

Both spellings of the inline value are accepted today, but the working group has been discussing which one the specification should keep. Use `display: grid-lanes` for anything you ship, and treat the inline form as less settled.

3.7 Project checkpoint: the Lanes Gallery

We can now explain every line in the layout that has been running since *Chapter 1*. Listing 3.6 is the Lanes Gallery's `layout.css` for this chapter. It lays out the same as Listing 1.3, but it spells the gap out as two longhand properties so that each one is visible.

Listing 3.6. The Lanes Gallery as a grid lane, with every line explained

```
.gallery {
  /* Lay the cards out in lanes. */
  display: grid-lanes;

  /* The grid axis: as many lanes of at least 180px as fit, sharing
     the free space equally between them. */
  grid-template-columns: repeat(auto-fill, minmax(180px, 1fr));

  /* The gap between lanes. */
  column-gap: 16px;

  /* The gap between cards within a lane (the stacking axis). */
  row-gap: 16px;
}
```

At 752 pixels this gives three 240-pixel lanes, and the nine cards land in lanes `[1, 5, 9]`, `[2, 4, 7]`, and `[3, 6, 8]`, exactly as in Figure 1.3. The finished file for this chapter is `examples/ch03/gallery/layout.css`.

Try the following experiments, and predict the result before you run each one:

1. Change `minmax(180px, 1fr)` to `minmax(300px, 1fr)`. How many lanes does an 800-pixel window get?
2. Change `row-gap` to `0`. What happens to the lane heights, and which cards move?
3. Delete `grid-template-columns` altogether. What do you get?

(The answers: two lanes; the cards in each lane pull together and the gallery gets shorter; and a single stack, as in §3.2.)

3.8 Summary

- `display: grid-lanes` makes a block-level grid lane container, and `display: inline-grid-lanes` makes an inline-level one.
- With no lanes defined, a grid lane is a single stack.
- Lanes are defined with `grid-template-columns` (column lanes) or `grid-template-rows` (row lanes), using everything you know about Grid's tracks.
- A grid lane has a **grid axis**, where it behaves like Grid, and a **stacking axis**, where it behaves like Flexbox. Items take their lane's size in the grid axis and their content's size in the stacking axis.
- The gap between lanes follows the grid axis, and the gap between items follows the stacking axis. For column lanes, that is `column-gap` and `row-gap`.
- The way a grid lane's orientation is chosen is still under discussion. Defining `grid-template-columns` or `grid-template-rows` works everywhere today.

Questions

1. What does a container with `display: grid-lanes` and no template do with its children?
2. What is the difference between the grid axis and the stacking axis, and which one controls an item's height in a column layout?
3. In a column layout, which gap property separates the lanes and which separates the cards inside a lane?
4. A container has both `grid-template-columns` and `grid-template-rows`. Which defines the lanes?

5. How wide is a container with `display: inline-grid-lanes` and `grid-template-columns: repeat(3, 120px)` with a 16-pixel gap?

The answers are in the *Assessments* appendix.

Further reading

- *Introducing CSS Grid Lanes*, WebKit blog: webkit.org/blog/17660
- *The Field Guide to CSS Grid Lanes*, WebKit: gridlanes.webkit.org
- *CSS Grid Layout Module Level 3*, W3C editor's draft: drafts.csswg.org/css-grid-3

About the Author

Sandeep Kumar Patel is a senior web developer and the founder of www.tutorialsavvy.com, a widely read programming blog since 2012. He has more than six years of experience in object-oriented JavaScript and JSON-based web application development. He qualified in GATE-2005 Information Technology (IT) and holds a master's degree from VIT University, Vellore.

End of the sample

This is the end of the free sample: the Preface and the first three chapters. The full book continues with:

- Chapter 4: How Items Get Packed
- Chapter 5: Taming the Flow
- Chapter 6: Choosing an Axis: Rows vs. Columns
- Chapter 7: Sharing Lanes with Subgrid
- Chapter 8: Building Responsive Masonry
- Chapter 9: The Reading Order Problem
- Chapter 10: Falling Back Gracefully
- Chapter 11: Gallery Patterns in Practice
- Chapter 12: Shipping the Gallery

Along with the questions and answers for every chapter, and the code for the whole Lanes Gallery project.