CSS Houdini in Practice

A tested guide to the browser's own drawing and styling APIs

Sandeep Kumar Patel

CSS Houdini in Practice

A tested guide to the browser's own drawing and styling APIs

Sandeep Kumar Patel

Edition: First edition, 2026.

Browser behaviour, version numbers, and specification status were verified on the dates stated in the text, and they will change. Each chapter that depends on a version explains how to re-check it.

Contents

Preface viii

About This Book x

Conventions Used in This Book xv

Technical Requirements xvii

About the Author xxi

Acknowledgments xxii

1. What Houdini Is 1

1.1 Following a Page Through the Browser 3

1.2 Meeting the Worklet 6

1.3 Meeting the Five APIs 7

1.4 Deciding What You Can Ship Today 10

1.5 Getting Gauge Running 12

1.6 Reading the Version Stamps 16

2. The `@property` Rule 19

2.1 Understanding Why Custom Properties Don't Animate 20

2.2 Registering a Property with `@property` 22

2.3 Choosing the Right Type 25

2.4 Animating a Gradient 27

2.5 Registering from JavaScript 30

2.6 Spotting a Registration That Failed 32

2.7 Registering Gauge's Theme Tokens 33

3. The Typed OM 41

3.1 Understanding the Problem with Strings 42
3.2 Creating Values That Know Their Unit 44
3.3 Reading Styles as Typed Values 45
3.4 Doing Arithmetic Without Parsing 47
3.5 Checking for the Typed OM 48
3.6 Building a Layout Readout for Gauge 50

4. Worklets 61

4.1 Loading Your First Worklet 62
4.2 Understanding What a Worklet Can't See 67
4.3 Sending the Worklet Data from CSS 70
4.4 Writing Gauge's First Painter 71
4.5 Keeping Worklets Away from the Bundler 74
4.6 Loading the Painter Only Where It Works 75

5. The Paint API 79

5.1 Using paint() as an Image 80
5.2 Drawing with the Canvas Commands 82
5.3 Feeding a Painter from CSS 87
5.4 Choosing a Transparent or Opaque Surface 91
5.5 Planning for Browsers Without the Paint API 91
5.6 Drawing a Chart from Data 94
5.7 Understanding When the Browser Repaints 101

6. Advanced Painting 105

6.1 Choosing How to Give the Ring Its Settings 106
6.2 Animating the Ring by Animating Its Input 108
6.3 Comparing a Painter with Plain CSS 116
6.4 Keeping the Ring Sharp on High-DPI Screens 120
6.5 Understanding What a Painter Can't Draw 120
6.6 Giving the Ring a Real Fallback 122
6.7 Debugging a Painter That Draws Nothing 128

7. The Layout API 134

7.1 Understanding What Custom Layout Would Do 135
7.2 Learning the Shape of a Layout Worklet 136
7.3 Building a Masonry Worklet 138
7.4 Getting the Masonry Shape Today with Columns 142
7.5 Understanding Why It Never Shipped 145
7.6 Switching to the Native Answer, `display: grid-lanes` 147

8. Animation Worklet 153

8.1 Understanding What Animation Worklet Promised 154
8.2 Learning the Shape of an Animator 156
8.3 Trying a Scroll-Linked Effect as a Worklet 157
8.4 Building the Same Effect with `scroll()` and `view()` 160
8.5 Finding the One Gap CSS Leaves 166
8.6 Choosing What to Ship 166

9. Going to Production 170

9.1 Putting Every Feature Check in One Place 171
9.2 Checking in CSS with `@supports` 174
9.3 Deciding Against the Paint API Polyfill 175
9.4 Planning for Browsers Without `@property` 176
9.5 Configuring Your Build for Worklets 177
9.6 Showing What This Browser Supports 178
9.7 Respecting High Contrast and Reduced Motion 182

10. Recipes with `@property` 189

10.1 Making a Number Count Up 191
10.2 Making a Gradient Move 197
10.3 Building a Ring with No Worklet 200
10.4 Giving Design Tokens Rules 205
10.5 Giving a Component a Setting 208
10.6 Knowing What Can't Be Registered 211

11. A Paint Worklet Cookbook 217

11.1 Writing a Painter That Behaves 219
11.2 Drawing Random Tiles That Survive a Resize 224
11.3 Drawing What a Gradient Can't 227
11.4 Making a Ripple from One Number 229
11.5 Checking What CSS Now Draws by Itself 233
11.6 Deciding When to Draw 234

12. Measuring What Houdini Costs 239

12.1 Knowing What You're Measuring: The Frame Budget 240
12.2 Recording What the Browser Does 242
12.3 Finding the Properties That Stay off the Main Thread 244
12.4 Trying to Time a Painter 246
12.5 Finding Where the Count Starts to Matter 247
12.6 Finding and Fixing the Unpainted Window 250
12.7 Reading the Numbers Honestly 253

13. Testing and Debugging 258

13.1 Testing a Painter by Reading Its Pixels 259
13.2 Testing the Fallback 265
13.3 Running the Tests in Three Engines 267
13.4 Proving the Tests Can Fail 269
13.5 Diagnosing Silent Failures 271
13.6 Guarding Every `@property` Rule with a Test 274

14. Houdini in Real Projects 280

14.1 Registering Tokens in One Place 281
14.2 Registering Properties for a Web Component 282
14.3 Sending the Fallback from the Server 285
14.4 Getting a Worklet Past a Content Security Policy 288
14.5 Making Sure Visitors Get Your New Worklet 290
14.6 Printing Gauge, and Fixing What Vanishes 291

15. Rolling It Out 298

15.1 Giving a Feature to a Few Visitors First 299
15.2 Finding Out What Visitors' Browsers Support 303
15.3 Adding Off Switches and Safe Defaults 305
15.4 Reporting Worklet Errors 309
15.5 Noticing When the Platform Changes 311
15.6 Writing the Decision Down 313

16. What's Next 318

16.1 Settled: `@property`. Two-Thirds There: The Typed OM 319
16.2 Shippable with Care: The Paint API 320
16.3 Not Yet: Layout, Animation, Parser, Font Metrics 321
16.4 What Native CSS Is Absorbing 322
16.5 Nearby, but Not Houdini 324
16.6 What to Do Now 324

A. API Reference (September 2026) 326

B. The Gauge App 331

C. Exercise Answers 336

Glossary 357

Index 364

Preface

CSS Houdini was announced around 2016 as a set of low-level APIs that would let you extend the browser's styling engine directly. A decade on, the results are uneven, and the unevenness is the point of this book. One of the five APIs shipped everywhere and is genuinely useful without qualification. A second is two-thirds there: solid in Chrome and Safari, absent in Firefox. A third shipped in Chromium only. The last two never left an experimental flag, while plain CSS grew features that cover most of what they promised.

This book is a guide to what works today, not a reference to a moving target. Its first nine chapters cover the five APIs themselves. The last seven cover what the tidy examples leave out: recipes that need no worklet, what a painter costs, how to test one, how to run it behind a server and a security policy, how to roll it out and watch it, and what to adopt. Every chapter is built on one small app, Gauge, an analytics panel. Where an example shows Gauge's own source, it is a verbatim excerpt, checked automatically against the real file before every build; where an example is illustrative or deliberately broken to show a failure, the text says so.

How This Book Was Checked

Every runnable example was actually run, in a real browser, before it was described as working. Every version number and support claim was checked against a primary source or a live browser on a stated date, not recalled. The research behind those claims is in a dated dossier included in the book's code bundle. The browsers were Chromium 153, Firefox 155 and WebKit 26.6, driven by the testing tool Playwright, plus real Safari 18.0, driven by Safari's own automation tool, for the places where Playwright's WebKit and Safari could differ. Safari 26 was not available to test.

This process caught the author's own first drafts being wrong more than once, and where it did, the correction is left visible in the text rather than quietly edited away. A first draft of the chapter on `@property` predicted that an invalid rule would log a console error; it does not, it fails silently, and that is now what the chapter teaches. A draft of the Paint API chapter assumed `paint()` could take arguments; it cannot, outside an experimental flag. Those corrections are more useful than the parts that were right the first time.

Everything in this book was checked in 2026. Browser support only moves in one direction for the shipped APIs, so assume `@property` and the Typed OM have only become more solid. For the rest, check the current status before you build on it. The last section of every relevant chapter tells you how.

About This Book

Who This Book Is For

Anyone who knows the basics of HTML, CSS and JavaScript: you can write a stylesheet, you have used a custom property such as `--brand-colour`, and you can read a JavaScript function. You do not need to know what a worklet is, how a browser draws a page, or anything about the Houdini specifications. The book explains each of those as it goes, and every new term is defined the first time it appears, and again in the Glossary.

This is a book you build along with. From Chapter 2 on, each chapter starts from its own copy of Gauge, the example app, in `code/steps/chapter-NN/`, and you add that chapter's part a few lines at a time, checking each step in the browser. You meet each problem before its answer: many sections start with the obvious code, let you watch it fail, and then fix it with you. If you are already experienced, the "Going deeper" boxes carry the extra detail, and you can skip the "Try it" boxes.

You do not need to be sold on the idea of extending the browser's engine, either. A fair amount of this book is about knowing when not

to. If you tried Houdini once in 2019, hit the browser-support wall, and left, this book is written with you in mind.

What This Book Is Not

It is not a specification reference, and it is not comprehensive. It is a guide to what works today, built on one small app, with every claim dated and every code example that shows Gauge's own source checked against the running code. Where an API does not work, the book says so and moves on rather than teaching it thoroughly for completeness' sake. Where one does, the book goes on to the parts a first tutorial skips: cost, tests, deployment, and what happens after release.

How This Book Is Organized

Chapters 1 to 3 are the portable parts. Chapter 1 lays out the rendering pipeline and where each of the five APIs hooks into it, and gets Gauge running on your machine. Chapter 2 covers `@property`, the one Houdini feature with no caveats. Chapter 3 covers the Typed OM, which is useful in Chrome and Safari and absent in Firefox.

Chapters 4 to 6 are the Paint API. Chapter 4 is the worklet mechanism on its own, so the drawing chapters do not each re-explain it. Chapter 5 introduces `paint()` and the fallback pattern that every use of it in this book follows. Chapter 6 goes deeper:

animating a painter, the no-data state, and the ceiling on what a painter can do.

Chapters 7 and 8 are the parts that never shipped. Both build the worklet anyway, run it behind a browser flag, and then show the native CSS feature that replaced it.

Chapter 9 is the first step towards production. It covers feature detection, fallbacks, and build configuration for everything the earlier chapters built.

Chapters 10 to 15 are the applied chapters. Chapter 10 is recipes that use only `@property` and need no worklet: counters, moving gradients, a ring meter, typed design tokens. Chapter 11 is a cookbook of painters, and of when plain CSS does the same job. Chapter 12 measures what all of it costs, on the main thread and at load. Chapter 13 tests it, in three engines, including how to tell whether a test can fail. Chapter 14 takes it into real projects: shadow DOM, server rendering, Content Security Policy, caching, and print. Chapter 15 is the rollout: a staged release, kill switches, reports from the field, and a written decision.

Chapter 16 is the verdict. It sums up what to adopt, what to skip, and what native CSS has already absorbed.

Reading Paths

If you only have an hour: Chapter 1 for the landscape, Chapter 2 for the one API worth adopting today, and Chapter 16 for the verdict.

If you are here to ship something: Chapters 1, 2, 5, and 9, then 12 to 15. That is `@property`, the Paint API, how to deploy both without breaking browsers that lack them, and how to measure, test, and roll them out.

If you want animation and theming without a worklet: Chapters 2 and 10. Everything in Chapter 10 runs in all three engines.

If you want the whole survey, including the dead ends: read it front to back. The chapters build on one another, and the later ones reuse what the earlier ones tested.

If you are evaluating whether Houdini is worth your time at all: Chapter 1 §1.4 answers that directly, early in the chapter, and Chapter 16 answers it again with the reasoning.

About the Running Example

Everything is demonstrated on one small app called **Gauge**: an analytics panel with stat cards, sparklines, ring meters, a theme switcher, and a "no data" state. It grows across the chapters and is deliberately tiny, because the subject is the APIs, not the app. Appendix B documents it in full.

Exercises and Answers

Every chapter opens with a list of what you will learn and a Technical Requirements section that says which browsers, flags and starting code you need. Inside, each section explains an idea before it shows the code, then walks through how the code works and what you should see when you run it. Chapters 2 to 16 close with a summary and a few exercises that run against Gauge. Appendix C gives the answer to each one, and every answer was run, in the browsers it names, before it went in. Try an exercise before you read its answer: several of them ask you to predict what will happen first, and a few of the predictions are wrong for the reasons the chapter warned about.

Conventions Used in This Book

The following typographical conventions are used in this book:

Bold
Indicates a new term where it is first defined (each one is also in the Glossary), and occasionally a rule worth remembering.

Italic
Used for emphasis.

`Constant width`
Used for code, and within paragraphs to refer to program elements such as CSS properties and values, JavaScript functions, file names, and anything you type: `@property`, `registerPaint`, `code/gauge/src/theme.css`. Section references such as §5.7 point to a numbered section.

> **TIP**
> This element signifies a tip or suggestion: a practical shortcut, or a small change to try in the example you have open.

> **NOTE**
>
> This element signifies a general note. Notes that go deeper into a mechanism can be skipped on a first read.

> **WARNING**
>
> This element indicates a warning or caution: a trap with a real cost, such as a silent failure, a value that throws, or a bundler that mangles a file.

How the Examples Are Checked

Examples that show Gauge's own source are excerpts of the real, running files, checked byte for byte against them by an automated test before each build, and so are the in-between versions you type on the way (`code/listings/`). A script runs every step of every chapter in three browser engines and checks that the page does what the text says. Where an example is illustrative only, or deliberately broken, the text says so, and each chapter from Chapter 2 on ends with "How We Checked This Chapter".

Every version number in this book is a photograph, not a fact: each claim says what it was checked against and when, and §1.6 explains how to get today's answer. Where a check caught an earlier draft being wrong, the correction is left visible.

Technical Requirements

What You Need

Requirement	Version	Why
Node.js	22 or later	To run the example app's dev server. (Node 20 reached end of life in April 2026.)
A Chromium browser	Chrome or Edge, current	Chapters 4 to 6 need the Paint API, which only Chromium has.
Firefox and Safari	current, optional	Useful for seeing the CSS fallbacks the book keeps insisting on. Not required.
A code editor	any	Anything that highlights JavaScript and CSS.

Check your Node version with `node --version` in a terminal. If it's below 22, or the command isn't found at all, install a current version from nodejs.org.

Chapters 1 to 3 work in any current browser. Chapters 4 to 6 need Chromium. Chapters 10 to 16 say at the start what each one needs.

Chapters 12, 13 and 15 also need Playwright: run `npm install` and then `npx playwright install` in `code/gauge/measure/` or `code/gauge/tests/`.

Browser Flags for Two Chapters

Chapters 7 and 8 cover APIs that have never shipped. To run their code at all you need to turn on flags, and the chapters tell you when:

- **Chapter 7 (Layout API)**: `chrome://flags/#enable-experimental-web-platform-features`, set to Enabled, then relaunch. The same flag also enables native `display: grid-lanes` in §7.6.
- **Chapter 8 (Animation Worklet)**: a deeper switch than the one above. Chrome must be launched with `--enable-blink-features=AnimationWorklet`. Chapter 8 explains why it is buried, and why even behind it the interesting case does not work.

Neither chapter's code is something you should ship. You can read both without turning on anything.

Using the Code Examples

The example app, **Gauge**, ships as a code bundle, `css-houdini-in-practice-code.zip` (80 KB), on Google Drive: download the code

bundle. If you are reading on paper, type this address into your browser:

```
https://drive.google.com/file/d/1I24WDRSO0vtJQsnVf-_m2E9Q_GL1PsyX
```

It opens the file's page on Google Drive; use Drive's **Download** button. There is no separate public repository: this zip is the complete code. It holds `code/gauge/`, the finished app; `code/steps/` and `code/listings/`, each chapter's starting copy and in-between files; and `research/`, the dated sources behind every support claim.

Run it with:

```
cd code/gauge
npm install
npm run dev
```

Then open the URL it prints. To build along with a chapter, run the same commands in its `code/steps/chapter-NN/` folder. Appendix B lists the app's query flags.

Every code example in this book that shows Gauge's source is an excerpt of the real file, verified against it automatically. If an example and the code bundle ever disagree, the code bundle is correct and the book has a bug: please report it (see "Errata").

Licence for the code: Gauge ships under the MIT licence (the full text is in `code/gauge/LICENSE` in the code bundle). Use it, modify it, or build on it in your own programs, free of charge, with no need to ask. **This covers the code only.** The book's own text is not code: reproducing a substantial portion of it, in this book's words, still needs permission.

Errata

Every technical claim in this book was checked on a stated date, and browsers move. Some of it will be wrong eventually, and some of it may be wrong now.

If you find an error, please report it to **sandeeppateltech@gmail.com**. Confirmed corrections are folded into the next revision.

About the Author

Sandeep Kumar Patel is a front-end developer with over 17 years of experience building for the web, working across HTML, CSS, JavaScript, Angular, React, Lit, and Web Components. He writes about the web platform's standards-track features and the gap between what they promise and what browsers actually ship at tutorialsavvy.com, and is the author of *Web Components and Design Systems* and *WebMCP in Practice*.

Acknowledgments

This book was written against real browsers, not documentation alone. So it owes a debt to the people who make browser internals inspectable: the Chromium, Gecko, and WebKit teams, the maintainers of MDN and caniuse.com, and the CSS Working Group members whose public discussions explain why the unshipped parts of Houdini stayed unshipped.

> **A disclosure, in the spirit of the rest of the book:** as of this edition, no independent technical reviewer has read this manuscript. Its claims were verified by running them in Chrome, Firefox, and Safari, and its code listings are machine-checked against the running example app, but no second pair of expert eyes has passed over the prose. That self-checking process caught dozens of things across several passes: a fabricated version number, a real accessibility bug in the shipped example app, and more than one claim that sounded right and had simply never been run. It is not a guarantee that it caught everything, and a self-review has a blind spot a second reader would not: if you are a CSS specialist willing to review a future revision, the errata address is in Technical Requirements.

CHAPTER 1

What Houdini Is

Most of the time, CSS gives you a fixed menu. You can choose any colour, any gradient, any grid, but only from the options the browser already knows. If you want a background the browser can't draw, or a layout it doesn't offer, you have to leave CSS behind: draw into a `<canvas>`, or move elements around with JavaScript.

CSS Houdini is a set of browser features that let you add to that menu. You write a little JavaScript, hand it to the browser, and the browser uses it as if it were part of CSS. It is named after Harry Houdini, the escape artist, because it lets you get inside the browser's styling engine, a box that is normally locked.

Houdini was announced around 2016 as five separate features, each opening a different part of the engine. Ten years later, they have had very different fates. One works in every browser and is worth learning today. One works in most browsers. One works only in Chrome and Edge. Two never shipped at all. This book teaches all five, and is honest about which ones to use.

This chapter gives you the map. You don't need to remember every detail; the chapters that follow explain each feature properly, one at a

time, as you build.

In this chapter, you will learn:

- the five steps a browser takes to turn your code into a picture on screen
- what a worklet is, and why it runs under such strict rules
- what each of the five Houdini features does, and where it works today
- how to check which features your own browser has
- how to get Gauge, the small app this book builds, running on your machine

Technical Requirements

- **Browser:** any current browser, for most of the chapter.
- **To run Gauge at the end:** Node.js 22 or later and the book's code bundle; section 1.5 explains how to download it.

From the next chapter on, you build: each chapter starts from a copy of Gauge as the previous chapter left it, explains a new idea, and then adds it to the app a few lines at a time, telling you what you should see after each step. What was tested, and in which browsers, is collected at the end of each chapter, under "How We Checked This Chapter".

1.1 Following a Page Through the Browser

To understand Houdini, you first need a rough picture of what a browser does when it shows a page. It works in five steps, always in the same order. This sequence is called the **rendering pipeline**.

Think of it like building and decorating a room:

1. **Parse.** The browser reads your HTML and CSS, which are just text, and turns them into data it can work with: a tree of elements (the **DOM**) and a list of style rules. This is reading the plans.
2. **Style.** For each element, the browser works out a value for every CSS property. It applies the cascade, works out inheritance, turns `2em` into pixels, and replaces each `var(--something)` with its value. The value it settles on at this step is called the **computed value**. (It isn't always the last word: a `width: auto` is only turned into a number of pixels later, by Layout.) This is choosing the finishes for each room.
3. **Layout.** Now the browser knows each element's styles, it works out sizes and positions: how wide each box is and where it sits. Flexbox and grid do their work here. This is measuring where the walls go.
4. **Paint.** The browser decides what colour every pixel should be: backgrounds, borders, text, shadows. This is painting the walls.

5. **Composite.** Some parts of the page are painted onto separate layers, like sheets of glass. The last step stacks those layers and puts the result on screen. Moving or fading a whole layer (with `transform` or `opacity`) can happen here without painting anything again, which is why those two properties are fast to animate.

Each step depends on the one before it. You can't lay out a box until you know its styles, and you can't paint it until you know its size.

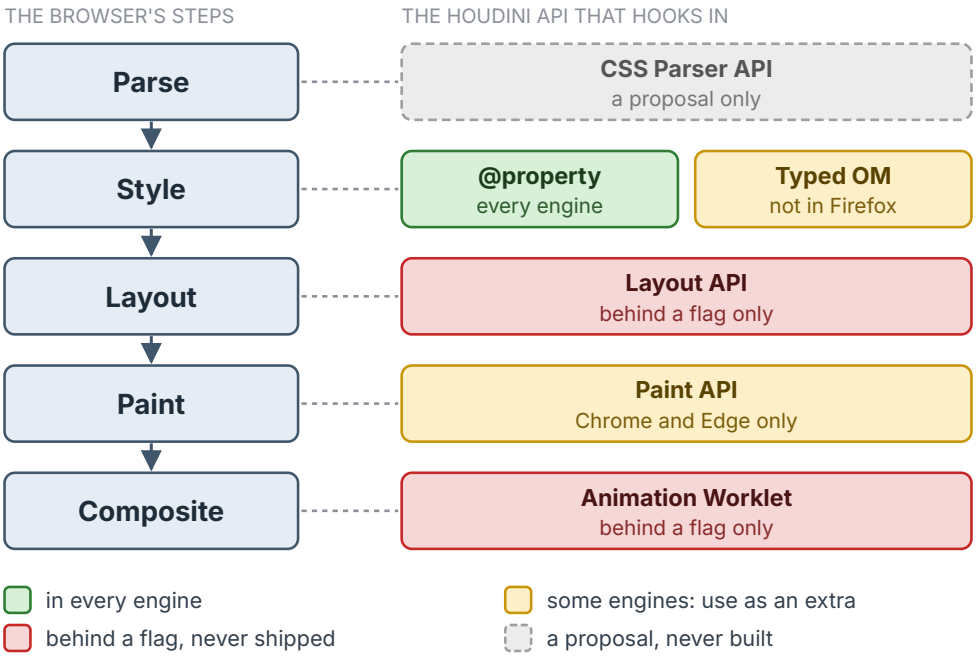

Figure 1-1. The rendering pipeline, with each Houdini API drawn as a hook onto the stage it extends.

Each Houdini feature attaches to one step. The official names are long, so here they are with what each one lets you do:

- **Properties and Values API**, used through `@property`: tells the browser what *type* of value a custom property holds, such as a colour or a length. Attaches to **Style**.
- **Typed Object Model** (**Typed OM**): lets JavaScript read and write CSS values as numbers and units, instead of strings. Also attaches to **Style**.
- **Paint API**: lets you write a function that draws an image, and use it in CSS anywhere an image goes, such as `background-image`. Attaches to **Paint**.
- **Layout API**: lets you write your own layout algorithm, a new kind of `display`. Attaches to **Layout**.
- **Animation Worklet**: lets you control an animation's progress from code, for example from the scroll position. Attaches near **Composite**.

Keep Figure 1-1 in mind. When the book says a painter "runs again", it means the Paint step runs again, and nothing before it has changed.

> **NOTE**
>
> Real browsers run some of these steps in parallel and on separate threads. But the order holds, and it matters: by the time Paint runs, Layout has finished, so a painter can read an element's size but can't change it.

1.2 Meeting the Worklet

Three of the five features (Paint, Layout and Animation Worklet) need you to hand some JavaScript to the browser's rendering engine. They all do it in the same way, with a **worklet**.

Before explaining a worklet, you need one more term. A web page's JavaScript normally runs on one thread, called the **main thread**. The same thread handles clicks, runs your scripts, and also does the Style, Layout and Paint steps. If your code keeps it busy, the page can't update, and animations stutter.

A **worklet** is a small JavaScript file that the browser runs *as part of rendering*, not as part of your page. Because rendering has to be fast and predictable, a worklet gets a very limited environment:

- It can't see the page. There is no `window`, no `document`, and no DOM.
- It can't use the network or storage. There is no `fetch` and no `localStorage`.
- It can't keep reliable state between calls. The browser may run several copies of it at once, and may throw a copy away and start a new one whenever it likes.

That sounds restrictive, and it is. Here is why. The browser might call your code hundreds of times a second, in any order, possibly on several threads at once. The only way to make that safe is to give the

code nothing it could break. A worklet takes some input, does its job, and has no other effects.

You load a worklet by giving the browser the file's URL:

```
CSS.paintWorklet.addModule('/worklets/sparkline.js');
```

Inside that file you write a class with one method the browser will call: `paint()` for the Paint API, `layout()` for the Layout API, and `animate()` for Animation Worklet. You never call that method yourself. The browser calls it when it needs to, and passes in what the method needs, including the values of any CSS properties the worklet has asked to read.

> **NOTE**
>
> A worklet is not a Web Worker. A Web Worker is a background thread you exchange messages with; a worklet is code the browser calls while rendering, and it can't send anything back to the page.

1.3 Meeting the Five APIs

Here are the five features, what each one is for, and where it works as this is written. An **experimental flag** is a hidden setting that a developer can switch on in their own browser to try something unfinished. Ordinary visitors never have it on, so a feature that only works behind a flag doesn't work for your users.

Feature	What it lets you do	Where it works	Advice
`@property` (Properties and Values API)	Give a custom property a type, a default and inheritance, which also lets it animate	Every current browser	**Use it now.** Chapter 2.
Typed OM	Read and write CSS values as numbers and units in JavaScript	Chrome, Edge, Safari. **Not Firefox.**	**Use it, with a fallback for Firefox.** Chapter 3.
Paint API	Draw an image with code and use it in CSS	Chrome and Edge only	**Use it for decoration**, with a plain-CSS version underneath. Chapters 4 to 6.
Layout API	Write your own layout algorithm	Nowhere, except behind a flag	**Learn it, then use `grid-lanes` or `columns`.** Chapter 7.
Animation Worklet	Drive an animation from code, such as from scrolling	Nowhere, except behind a flag	**Learn it, then use scroll-driven animations.** Chapter 8.

Two more Houdini ideas, the **CSS Parser API** and the **Font Metrics API**, never got beyond a proposal. No browser has built

either.

Check Your Own Browser

You don't have to take the table on trust. Open any ordinary web page, or a local HTML file, then open the browser's developer tools (F12, or Cmd+Option+I on a Mac; in Safari, turn on the Develop menu in Settings first) and go to the **Console** tab. Paste this and press Enter:

```
console.table({
  '@property': 'registerProperty' in CSS,
  'Typed OM': 'computedStyleMap' in Element.prototype,
  'Paint API': 'paintWorklet' in CSS,
  'Layout API': 'layoutWorklet' in CSS,
  'Animation Worklet': 'animationWorklet' in CSS,
});
```

Each line asks whether the browser has one of the objects that feature adds, and answers `true` or `false`. Here is what three browsers answered:

Feature	Chrome 153	Firefox 155	WebKit 26.6 (Safari's engine)
`@property`	true	true	true
Typed OM	true	false	true
Paint API	true	false	false
Layout API	false	false	false

Feature	Chrome 153	Firefox 155	WebKit 26.6 (Safari's engine)
Animation Worklet	false	false	false

This is a quick check, not a perfect one. A browser can have an object without the feature fully working, so the book builds a more careful test later. It is enough to see the pattern for yourself. (One small trap: on a blank tab, `about:blank`, Chrome answers `false` for the Paint API. Use a real page.)

1.4 Deciding What You Can Ship Today

"Ship" means putting it in front of real users. Here is what the table means for that.

`@property` **is safe everywhere.** Every current browser supports it, with no flag. It is the one Houdini feature this book recommends with no conditions, and the next chapter starts with it.

The Typed OM is missing in Firefox. Firefox hasn't built it, and hasn't said when it will. So whenever this book uses the Typed OM, it also shows the older way of doing the same thing with strings, which works everywhere.

The Paint API works only in Chrome and Edge, and that is unlikely to change. Firefox never built it. Safari has code for it, but

has kept it switched off for years. There used to be a **polyfill**, a script that adds a missing feature to browsers that lack it, but it was retired in April 2026. So this book follows one strict rule: every painted effect sits on top of a plain-CSS version that works everywhere. Users of Chrome and Edge get the painted version, and everyone else gets the plain one. This approach, building something that works for everyone and then adding extras where the browser supports them, is called **progressive enhancement**.

The Layout API and Animation Worklet aren't usable, and may never be. Neither has left an experimental flag in any browser, after many years. Meanwhile, plain CSS has caught up with the main things people wanted them for:

- People wanted Animation Worklet mostly to link animations to scrolling. CSS can now do that directly, with `animation-timeline: scroll()`.
- People wanted the Layout API mostly for **masonry**, the Pinterest-style layout where items of different heights pack into columns. CSS now has that too, as `display: grid-lanes`, first shipped in Safari 26.4.

The book still builds both worklets. Seeing them run behind a flag explains what these APIs were for, and why the plain CSS versions look the way they do. Then they show you the CSS feature to use instead.

> **WARNING**
>
> Browser support changes. Every version number in this book was checked on a stated date, and §1.6 explains how to check it again yourself.

1.5 Getting Gauge Running

Every example in this book runs in one small app called **Gauge**. It is an analytics panel, the kind of thing you might see in the corner of a dashboard: a row of cards, each showing one number, whether it is going up or down, and a small line chart.

Gauge is kept deliberately small so that the only interesting code in it is the Houdini code. It has no framework, no chart library and no server. The rest of its code fits in Appendix B.

Figure 1-2, a little further on, shows Gauge as it looks when every chapter's work is finished, with the pieces the book builds numbered. This table says what each number is, and which chapter builds it:

No.	What it is	Built with	Chapter
1	The theme switch, where the colours glide from blue to orange	`@property`	2

No.	What it is	Built with	Chapter
(not shown)	A developer-only overlay that shows each card's measured size	Typed OM	3
2	The small coloured wedge in the corner of each card	a first paint worklet	4
3 and 4	The ruled background of each card, and the sparkline chart	Paint API	5
5 and 6	The ring meter that sweeps round, and the hatching on the card with no data	Paint API and `@property`	6
(a flag)	The cards arranged as masonry	Layout API, then `grid-lanes`	7
(a flag)	A shadow under the header that appears as you scroll	Animation Worklet, then `scroll()`	8

Read the table from top to bottom and you have the order of the book: the features that work everywhere first, then the Chrome-only ones, then the ones that never shipped.

Running Gauge

Gauge is in the book's code bundle, a zip file on Google Drive: download the code bundle. On paper, the address to type is:

```
https://drive.google.com/file/d/1I24WDRSO0vtJQsnVf-_m2E9Q_GL1PsyX
```

Technical Requirements, at the front of the book, has the same link. Unzip the file, and Gauge is in the `code/gauge/` folder. It needs **Node.js**, the program that runs JavaScript outside a browser, version 22 or later. If you don't have it, install it from nodejs.org. Then open a terminal in the folder where you unzipped the code bundle and run:

```
cd code/gauge
npm install
npm run dev
```

`npm install` downloads the one tool Gauge depends on, **Vite**, a small development server. `npm run dev` starts it, and prints an address, `http://localhost:5199/`. Open that address in Chrome or Edge. If the terminal says `Port 5199 is already in use`, another copy of the server is still running: stop it with Ctrl+C in its terminal and try again.

You should see the page in Figure 1-2. In Firefox or Safari you will see the same cards without the painted parts: plain backgrounds, a

flat strip where the sparkline would be, and rings drawn with SVG instead. That is the plain-CSS version doing its job.

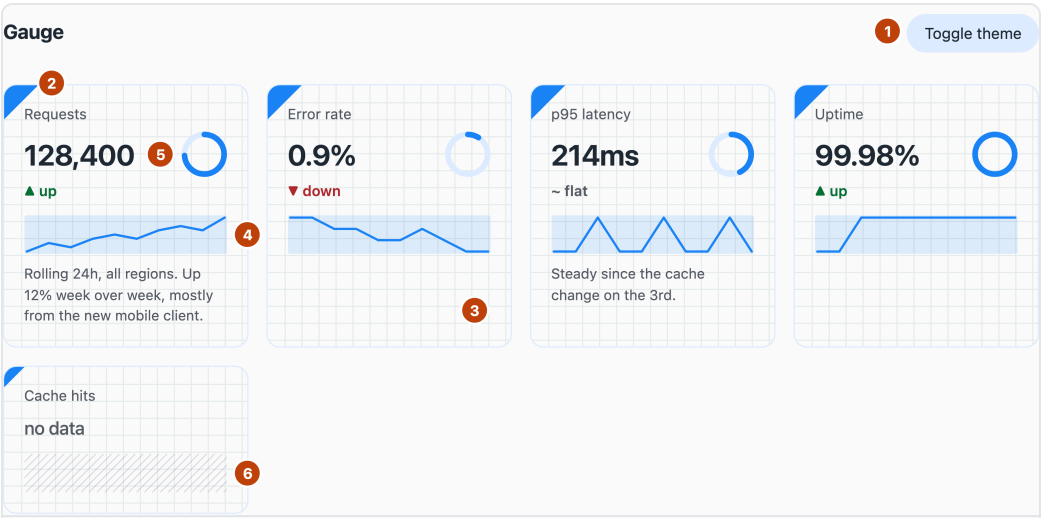

Figure 1-2. Gauge, finished. The numbers mark the parts the book builds: (1) the theme switch, (2) the corner wedge, (3) the ruled background, (4) the sparkline, (5) the ring meter, and (6) the "no data" hatching.

Two things might look different from the figure:

- **Colours.** If your computer is set to dark mode, Gauge uses its dark colours. Every figure in this book was taken in light mode.
- **Everything is already there.** `code/gauge/` is the *finished* app, so you can see where the book is going.

You won't build in that folder, though. From Chapter 2 on, each chapter starts from its own copy of Gauge, `code/steps/chapter-02/`, `code/steps/chapter-03/` and so on: Gauge exactly as the chapters before it leave it, without the part that chapter adds. You

open that folder, run the same two commands (`npm install` , then `npm run dev`) and build the missing part yourself, step by step, checking each step in the browser. When you finish a chapter, your copy matches the next chapter's starting point, so you can carry on in your own copy or switch to the next folder. Either way, if something goes wrong, you can always pick up cleanly at the next chapter.

Only one of these servers can use port 5199 at a time, so stop one (Ctrl+C) before starting another.

1.6 Reading the Version Stamps

Browser support keeps changing, but a book can't. So this book never states a version number as a permanent fact. When a claim depends on a browser version, the book says which version was tested and when, so you can tell how old the claim is and check it again.

Three ways to check:

- **Support tables.** caniuse.com shows which browsers support a feature. For `@property` , the page is `caniuse.com/mdn-css_at-rules_property` . For the Paint API it is `caniuse.com/css-paint-api` , and for the Typed OM, `caniuse.com/mdn-api_stylepropertymap` .
- **Your own browser.** The console check earlier in this chapter takes ten seconds, and the book builds a more careful version later.

- **The specifications.** The official documents that define each feature are at drafts.css-houdini.org.

This is when the numbers in this chapter were checked:

Feature	First shipped in	Share of users (approximate)
`@property`	Chrome and Edge 85, Firefox 128, Safari 16.4	about 95%
Typed OM	Chrome 66, Edge 79, Safari 16.4	Firefox has none
Paint API	Chrome 65, Edge 79	about 78%

The table comes from the browser makers' release notes (2026-09-09), checked again in Chrome 153, Firefox 155 and WebKit 26.6. The shares come from caniuse and change often, so look up today's.

> **NOTE**
>
> Every support claim in this book comes from a dated research file, `research/houdini-facts-2026-09.md` in the code bundle, which lists its sources.

If you are reading this a year or more after 2026, assume `@property` is still fine. Then check three things:

- whether Firefox has added the Typed OM;

- whether Firefox now supports scroll-driven animations;
- whether the Layout API or Animation Worklet has finally moved.

The rest of the book assumes they haven't.

Summary

In this chapter, you learned that CSS Houdini is five browser features that hook your JavaScript into the rendering pipeline (Figure 1-1). Three of them hand your code to the browser as a worklet, a small file the browser runs during rendering. `@property` works everywhere, the Typed OM everywhere except Firefox, and the Paint API only in Chrome and Edge; the Layout API and Animation Worklet never shipped. So the advice comes in three kinds: use it, use it as an extra on top, or use the plain CSS that replaced it.

In the next chapter, you'll start building Gauge with `@property`.

CHAPTER 2

The `@property` Rule

If you have used CSS **custom properties**, the ones that start with two dashes, like `--brand-colour`, you know how useful they are for keeping a value in one place. They have one limitation that most people never notice until they try it: you can't animate them. Put a custom property under a `transition` and the value jumps instead of gliding.

The `@property` rule removes that limitation, and it is the one Houdini feature that works in every current browser. In this chapter, you'll use it to make Gauge's theme switch fade smoothly from one set of colours to another.

In this chapter, you will learn:

- why the browser can't animate an ordinary custom property
- how to register a property with `@property`, and what each part of the rule does
- how to choose the right type for a property
- how to animate a gradient by animating one value inside it
- how to register a property from JavaScript instead, and how that differs

- how to spot the one mistake that makes a registration fail silently

Technical Requirements

- **Browser and editor:** any current browser and a text editor. The first half of the chapter is a small page of your own.
- **Starting code:** `code/steps/chapter-02/`, where the theme switch changes colours instantly.
- **Setup:** open a terminal in that folder, run `npm install` once and then `npm run dev`, and open the address it prints.

2.1 Understanding Why Custom Properties Don't Animate

To animate from one value to another, the browser has to work out all the values in between. Going from `10%` to `90%` over one second, it needs `11%`, then `12%`, then `13%`: one new value for every frame it draws. Working out those in-between values is called **interpolation**.

The browser can interpolate a number, because halfway between 10 and 90 is 50. It can interpolate a colour, because it knows how to mix two colours. But it can't interpolate two pieces of text: there is no sensible answer to "what is halfway between `cat` and `dog`?"

And that is the problem, because an ordinary custom property **is** text. When you write `--fill: 10%`, the browser doesn't store "ten percent". It stores the characters `1`, `0` and `%`, and only makes sense of them later, when `var(--fill)` places them inside another property. Asked to animate `--fill` from `10%` to `90%`, it sees two bits of text, finds nothing in between, and swaps one for the other. The value jumps.

Being text has two more consequences:

- **Nothing checks the value.** Write `--fill: potato` and the browser accepts it without complaint. The mistake only shows up later, somewhere else, when the property that uses `var(--fill)` can't read `potato` and gives up.
- **A missing value has no useful default.** If `--fill` is never set, anything that uses `var(--fill)` behaves as if you had written `unset`, which is rarely what you meant.

See the jump for yourself. Create a file called `bar.html`, paste this into it, and open it in your browser:

```
<!doctype html>
<style>
  .bar {
    width: 300px;
    height: 40px;
    --fill: 10%;
    background: linear-gradient(90deg, royalblue var(--fill),
        #e6e9ef var(--fill));
    transition: --fill 1s linear;
```

```
  }
  .bar:hover {
    --fill: 90%;
  }
</style>
<div class="bar"></div>
```

How the code works: the bar's background is a gradient that is blue up to `var(--fill)` and grey after it. `--fill` starts at `10%`, the `:hover` rule changes it to `90%`, and the `transition` line asks the browser to take one second to get there.

You should see a grey bar with a blue section at the left. Now move your mouse over it. The blue doesn't slide across: it **jumps** straight to 90%. Nothing is misspelt, and it isn't a browser bug. It is the text problem.

2.2 Registering a Property with `@property`

The fix is to tell the browser what type of value the property holds. This is called **registering** the property, and the `@property` rule does it with three parts, called **descriptors**:

- `syntax` is the type, such as `"<percentage>"`. It is the important one: once the browser knows `10%` and `90%` are percentages, it can work out the values between them.
- `inherits` says whether a child element automatically gets its parent's value, the way `color` passes down. There is no default: you must write `true` or `false`.

- `initial-value` is the value the property has when nothing sets it. It must be a valid value of the type in `syntax`.

All three are required; the only exception is the special type `"*"`, described below. If any one is missing or wrong, the browser ignores the whole rule, without any error message.

Now register `--fill`. Add this rule at the top of the `<style>` block in `bar.html`:

```
@property --fill {
  syntax: "<percentage>";
  inherits: false;
  initial-value: 10%;
}
```

Save, reload the page, and hover over the bar again. This time the blue **slides** across, taking a full second.

> **TIP**
>
> In `bar.html`, change `"<percentage>"` to `"<color>"` and reload. Now the rule says `--fill` is a colour, but its initial value, `10%`, isn't one. So the rule is invalid, the browser throws it away, and the bar jumps again. Change it back.

Registering also fixes the other two problems of plain text:

- **The value is checked.** Set `--fill: potato` on an element and the browser rejects it, because `potato` isn't a percentage. The element gets the initial value, `10%`, instead.
- **An unset property has a proper default.** On an element that never sets `--fill`, `var(--fill)` gives `10%`, not nothing.

> **NOTE**
>
> Once a property is registered, it always has a value, its initial value. So the fallback in `var(--q, 50px)` is never used, and an invalid value falls back to the initial value, not to an earlier rule.

Choosing Whether a Property Inherits

Whether a property should inherit depends on what it describes. A simple rule of thumb: if the value describes the whole page, like a theme, let it inherit; if it describes one component, like one ring's progress, don't. Gauge has both kinds.

Its theme colours inherit. The accent colour is set once, on the page's root element, and every element in the page should see it:

```
@property --gauge-accent {
  syntax: "<color>";
  inherits: true;
  initial-value: oklch(62% 0.19 255);
}
```

`oklch()` writes a colour as lightness, chroma (how intense it is) and hue. You could write the same colour in hex; a Note later in this chapter explains why Gauge doesn't.

Its ring meters don't inherit. Each ring on the page shows its own value, so each keeps its own angle, and setting the angle on the whole panel doesn't leak into every ring inside it:

```
@property --gauge-ring-angle {
  syntax: "<angle>";
  inherits: false;
  initial-value: 0deg;
}
```

> **NOTE**
>
> A registration applies to the whole page, wherever the rule appears. If two `@property` rules use the same name, the last one wins.

2.3 Choosing the Right Type

The `syntax` descriptor is written in a small language of its own. Each type goes in angle brackets, inside quotes. These are the types you will use most:

`syntax`	**Accepts values like**
`"<length>"`	`10px`, `2rem`, `0`

`syntax`	**Accepts values like**
`"<number>"`	`1`, `0.5`, `-3`
`"<percentage>"`	`50%`
`"<length-percentage>"`	a length or a percentage, or a `calc()` mixing them
`"<color>"`	`red`, `#a0f`, `oklch(62% 0.19 255)`
`"<angle>"`	`90deg`, `0.25turn`
`"<time>"`	`200ms`, `1s`
`"<integer>"`	`1`, `-4` (whole numbers only)
`"<image>"`	`url(...)`, `linear-gradient(...)`, and `paint(...)` where the Paint API exists
`"<custom-ident>"`	any name you make up, such as `compact`

You can also combine types.

Lists. Put `+` after a type for a list separated by spaces, or `#` for a list separated by commas:

- `"<length>+"` accepts `4px 8px 12px`
- `"<color>#"` accepts `red, green, blue`

Choices. Use `|` to mean "or":

- `"<length> | auto"` accepts a length, or the word `auto`

- `"small | medium | large"` accepts exactly one of those three words, and nothing else

Anything. The special syntax `"*"` accepts any value at all, just like a custom property you never registered. The difference is that you can still choose `inherits`. A property registered with `"*"` can't be animated, and for this one syntax, `initial-value` is optional. Use it only when a value really has no fixed type.

> **TIP**
>
> Choose the narrowest type that fits, such as `"<length>"` for a size. The narrower the type, the more mistakes the browser catches. For a count that animates, use `<number>` and round it, because WebKit doesn't round an animating `<integer>`.

2.4 Animating a Gradient

CSS can't animate one gradient into another. It has no rule for working out what lies between two `linear-gradient(...)` values, so a `transition` on `background` doesn't run at all: the colour jumps. But a registered colour *can* be interpolated. So the trick is to move the part of the gradient that changes into a registered custom property, and let the gradient read it. The browser then animates the property, and on every frame draws the gradient again with the property's current value. The gradient itself is never animated. **One**

of its inputs is. That is exactly what happened with `--fill` in the progress bar.

To see the difference, try this in a new page. First, the version that doesn't work:

```
.panel {
  background: linear-gradient(90deg, blue, white);
  transition: background 400ms;
}
.panel:hover {
  background: linear-gradient(90deg, orange, white);
}
```

Hover over the panel and the colour changes, but it jumps instead of fading, in every browser. Now the version with the changing colour moved into a registered property:

```
@property --edge {
  syntax: "<color>";
  inherits: false;
  initial-value: blue;
}
.panel {
  background: linear-gradient(90deg, var(--edge), white);
  transition: --edge 400ms;
}
.panel:hover { --edge: orange; }
```

On hover, the browser animates `--edge` from blue to orange, and the gradient follows it smoothly.

Applying It to Gauge's Theme Switch

Gauge's theme switch can use the same trick, on a whole page at once. Its colours are four custom properties on the root element, and clicking **Toggle theme** adds the class `theme-warm`, which gives them their warm values. To make that fade, the four properties need a `transition`, and they need to be registered colours. You'll do it in that order, so you can see why both are needed.

Open `src/theme.css` in your Chapter 2 copy and add a `transition` to the `:root` rule, listing all four, so that it reads:

```
:root {
  --gauge-accent: oklch(62% 0.19 255);
  --gauge-accent-weak: oklch(94% 0.03 255);
  --gauge-surface: oklch(99% 0.004 255);
  --gauge-ink: oklch(28% 0.02 255);

  /* one place the theme transition is declared; every token rides
      it */
  transition:
    --gauge-accent 250ms ease,
    --gauge-accent-weak 250ms ease,
    --gauge-surface 250ms ease,
    --gauge-ink 250ms ease;
}

.theme-warm {
  --gauge-accent: oklch(66% 0.17 40);
  --gauge-accent-weak: oklch(93% 0.04 60);
  --gauge-surface: oklch(99% 0.008 70);
  --gauge-ink: oklch(30% 0.03 40);
}
```

Save, reload and click **Toggle theme**. It still snaps: the four colours aren't registered yet, so to the browser they are text, just like `--fill` before you registered it. You'll register them in the last section of this chapter. Once you do, this is what will happen when you click:

1. Gauge's script adds the class `theme-warm` to the root element.
2. That changes the four colour properties to their warm values.
3. Because they are registered colours, the `transition` on `:root` animates each one over 250 milliseconds.
4. Everything that uses them, such as a card border or the button, picks up the in-between colour on every frame.

> **NOTE**
>
> Colours written as `oklch()` blend evenly, so blue to orange passes through a clean mix. Written in hex or `rgb()`, the middle of the fade goes dull. That is why Gauge uses `oklch()`.

2.5 Registering from JavaScript

You can also register a property from JavaScript, with `CSS.registerProperty()`. It does the same job as the `@property` rule, with the same three descriptors; only the names are written in JavaScript style, so `initial-value` becomes `initialValue`:

```
CSS.registerProperty({
  name: '--gauge-accent',
  syntax: '<color>',
```

```
  inherits: true,
  initialValue: 'oklch(62% 0.19 255)',
});
```

There are three differences worth knowing:

- **It tells you when something is wrong.** A broken `@property` rule is ignored without a word. A broken `CSS.registerProperty()` call throws an error you can see: a `SyntaxError` if `initialValue` doesn't match `syntax` or is missing, and a `TypeError` if `inherits` is missing.
- **You can only call it once per name.** A second call for the same name throws an `InvalidModificationError`. If your code might run twice, check first or wrap the call in `try` / `catch`.
- **It beats an `@property` rule for the same name.** If both register the same name, the JavaScript version counts, whichever ran first. A forgotten `registerProperty()` call can therefore quietly override the `@property` rule you are looking at. Pick one way per property.

Use the CSS form by default: it lives with the rest of your styles and needs no script. Use the JavaScript form when the registration has to happen in code, for example in a component library that ships JavaScript but no stylesheet, or when the initial value is only known at run time.

2.6 Spotting a Registration That Failed

The most important thing to know about `@property` is how it fails. **If any descriptor is wrong, the browser throws away the whole rule, silently.** There is no message in the console. The property simply becomes an ordinary custom property again: text, not checked, not animatable.

The commonest cause is an `initial-value` that doesn't match its own `syntax`. Here is a rule with that mistake:

```
/* broken on purpose, do not copy */
@property --card-pad {
  syntax: "<length>";
  inherits: false;
  initial-value: 10;   /* 10 is a <number>, not a <length>: it
      needs a unit */
}
```

Before reading on, decide what you think happens. Most people expect a console error, or that the browser uses some default length. Neither happens: the rule is gone, and every problem of plain text is back, with nothing to tell you why. Leaving out `inherits`, or leaving out `initial-value` for any `syntax` other than `"*"`, has the same effect.

You can see it in `bar.html`. Change `initial-value: 10%` to `initial-value: 10`, reload, and hover. The bar jumps, exactly as it did before you added the rule.

Checking That a Registration Worked

Because the failure is silent, you need a way to check. The simplest is to read the property on an element that never sets it:

```
getComputedStyle(document.body).getPropertyValue('--fill');
```

If the rule was accepted, you get the initial value, `"10%"`. If it was thrown away, you get an empty string, `""`, because an unregistered property that nobody set has no value. Try it in `bar.html` with the good rule and with the broken one. Later in the book, you'll turn this check into an automated test for a whole stylesheet.

> **WARNING**
>
> A broken `@property` rule looks like a working one until something fails to animate. When a registered property behaves like an unregistered one, check the rule first: does `initial-value` match `syntax`, and is `inherits` there?

2.7 Registering Gauge's Theme Tokens

Now finish Gauge's theme switch. Gauge calls its colours **design tokens**: named values, such as the accent colour, that the whole design is built from. Registering them gives them a type, so the

`transition` you added can interpolate them. At the top of `src/theme.css`, replace the opening comment with this block:

```
/* Gauge's theme tokens. The Chapter 2 material.
 *
 * Registering these with @property does two things a plain `--x` can't:
 *   1. the browser validates the value (a bad --gauge-accent is rejected,
 *      not inherited as a broken string);
 *   2. the value becomes animatable: switching .theme-warm transitions the
 *      accent colour instead of snapping, because it's typed as <color>.
 */

@property --gauge-accent {
  syntax: "<color>";
  inherits: true;
  initial-value: oklch(62% 0.19 255);
}

@property --gauge-accent-weak {
  syntax: "<color>";
  inherits: true;
  initial-value: oklch(94% 0.03 255);
}

@property --gauge-surface {
  syntax: "<color>";
  inherits: true;
  initial-value: oklch(99% 0.004 255);
}

@property --gauge-ink {
  syntax: "<color>";
  inherits: true;
  initial-value: oklch(28% 0.02 255);
```

```
}

/* Not inherited: each ring owns its own angle. Animated in Chapter
    6. */
@property --gauge-ring-angle {
  syntax: "<angle>";
  inherits: false;
  initial-value: 0deg;
}
```

How the code works:

- The four colours use `syntax: "<color>"` and `inherits: true`, because a theme belongs to the whole page.
- The ring angle uses `"<angle>"` and `inherits: false`, because each ring has its own. Gauge has no rings yet; registering the angle now keeps every token in one place.
- Every `initial-value` is a complete, valid value of its type, so none of these rules is silently thrown away.

Save, reload and click **Toggle theme**. This time the whole panel fades from blue to warm orange in a quarter of a second (Figures 2-1 and 2-2). The figures in this book show the finished Gauge, so they include parts, such as the rings and sparklines, that you'll build in later chapters.

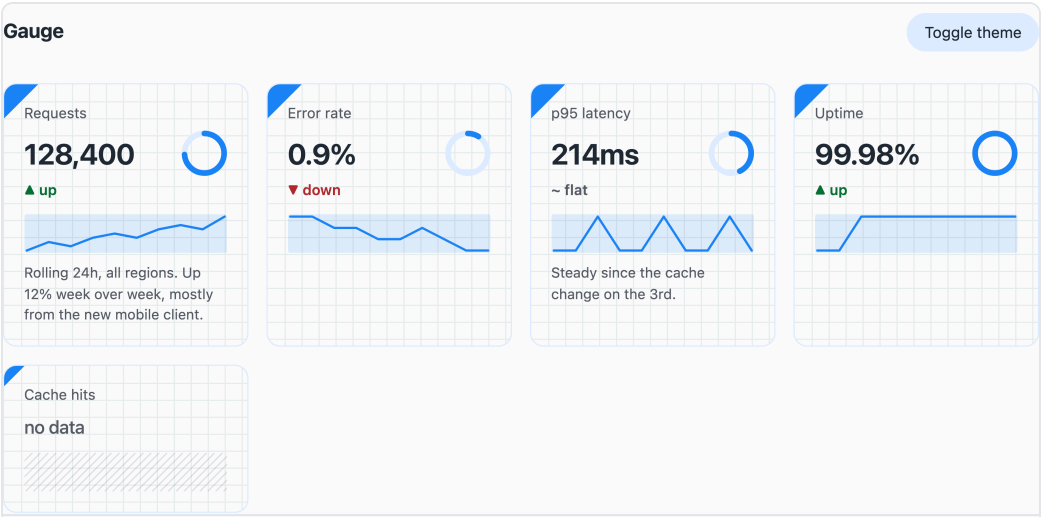

Figure 2-1. Gauge in its default theme: blue accent on the sparklines, borders, and button.

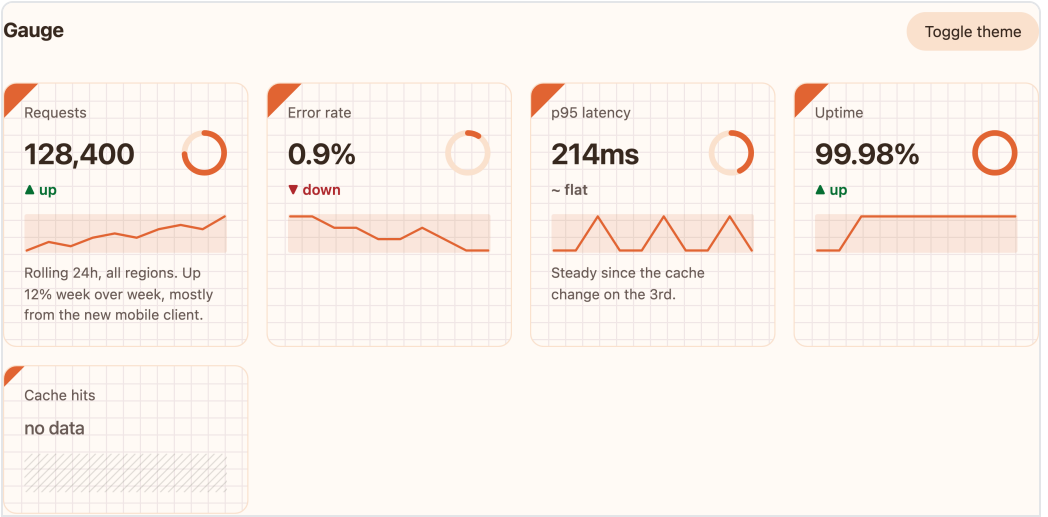

Figure 2-2. The same panel after Toggle theme, with every registered `<color>` token eased to its warm value.

`theme.css` also has a block, not shown above, that sets the same four colours for dark mode, so Gauge follows your operating system's

setting. If your computer uses dark mode, your Gauge will look different from the figures but behave the same way.

Watching the In-Between Colours

To see the in-between colours, open the console and paste this. It clicks the button for you, then prints the value of `--gauge-accent` on every frame for 300 milliseconds:

```
const root = document.documentElement;
const t0 = performance.now();
document.querySelector('.gauge-theme-toggle').click();
(function log() {
  console.log(getComputedStyle(root).getPropertyValue('--gauge-
      accent'));
  if (performance.now() - t0 < 300) requestAnimationFrame(log);
})();
```

You should see a list of slightly different colours, one per frame, ending at the warm orange. Now reload with `?notransition` on the end of the address, which turns every transition off, and run it again. This time you get one value: a jump. The browser can make that list of colours only because the `@property` rules told it these properties *are* colours.

Your copy of Gauge now matches `code/steps/chapter-03/`, where the next chapter starts.

Summary

In this chapter, you learned that an ordinary custom property is stored as text, so the browser can't check it, can't give it a useful default, and can't animate it: there is nothing between two pieces of text. Animating means interpolating, and interpolating needs a type. `@property` supplies one, with a `syntax`, an `inherits` flag and an `initial-value`, all three required. Once a property is registered it animates, which is how you animate a gradient: not the gradient itself, but one value inside it.

You saw that the `syntax` language covers the everyday types, lists with `+` and `#`, choices with `|`, and `*` for anything, and that `CSS.registerProperty()` does the same job from JavaScript and throws when something is wrong. Above all, you saw the silent failure: a single bad descriptor deletes the whole registration, so check one by reading the property on an element that doesn't set it. Of all the Houdini features, `@property` needs the least work and pays off most often: no worklet, no flag, no fallback to design, and it works in every current browser.

In the next chapter, you'll move from CSS to JavaScript, and read and write CSS values as numbers instead of text with the Typed OM.

Exercises

Each exercise runs against Gauge. Answers, with what each one was checked against, are in Appendix C.

Exercise 2-1. A token that glides. Register a fifth token in `theme.css`, `--gauge-radius`, as a `<length>` that inherits, with an `initial-value` of `12px`. Use it as the card's `border-radius` in `gauge.css`, give `.theme-warm` a value of `4px`, and add the token to the transition list on `:root`. Toggle the theme. Then delete only the `@property` rule and toggle again. How many different radii does a card pass through in each case, and how did you count them?

Exercise 2-2. Predict, then break it. Change `--gauge-accent`'s `initial-value` to `62% 0.19 255`, with the `oklch()` wrapper missing. Before you reload, write down what you expect: a console error, a fallback colour, or something else. Then check what the theme switch does and what the console says.

How We Checked This Chapter

Every page and step in this chapter is a real file (`code/listings/chapter-02/`, or the next chapter's starting point), and a script ran them in Chromium 153, Firefox 155 and WebKit 26.6, reading the animated value on every frame.

- **2.1 and 2.2.** Hovering `bar.html` without the rule gave exactly one value of `--fill`, `90%`, in all three engines. With the rule, Chromium gave 62 values in the second, from `10%`, `11.33%`, `12.66%` to `90%`; WebKit 61; Firefox, which drew twice as many frames,
  122. An element that never set `--fill` read `10%`. Real Safari 18.0 matched.

- **2.4.** The gradient-to-gradient transition gave one value from the first frame in all three engines and in real Safari (at about 24 milliseconds). The registered `--edge` version gave 23 to 46 distinct gradients. The midpoint of the `oklch()` blue and orange, sampled with a paused animation, was `oklab(0.64 0.040526 -0.037126)` in all three.
- **2.5.** The `SyntaxError`, `TypeError` and `InvalidModificationError` cases, and the JavaScript registration beating `@property` with the script run before and after the stylesheet, were tested in all three; the specification agrees.
- **2.6.** With `"<color>"` or `initial-value: 10`, the bar jumped (one value) and `--fill` on an element that never set it read `""` in all three engines.
- **2.7.** In Gauge, the toggle gave one accent value at the start of the chapter, still one with the `transition` added, 17 once the tokens were registered, and one with `?notransition`. The final copy matched `code/steps/chapter-03/` file for file.

That's the sample

Chapters 1 and 2 are the landscape and the one API worth adopting this week. The other fourteen chapters cover the Typed OM, three chapters on the Paint API, both of the APIs that never shipped (built, run, and honestly retired in favour of native CSS), production feature-detection, six applied chapters on recipes, cost, testing, deployment, and rollout, and Chapter 16's verdict tying all five APIs together.

Every code listing in the full book is checked automatically against the real, running example app. Every browser-support claim is dated and traced to a primary source. Get the rest of the book to see where each of the five APIs actually stands, not where a five-year-old blog post says it does.

(Where to buy: link to be added once the book is listed.)