

DevTools Timeline: Now Providing the Full Story



By Heather Mahan

Heather Mahan is a developer.

The DevTools Timeline panel has always been the best first stop on the path to performance optimization. This centralized overview of your app's activity helps you analyze where time is spent on loading, scripting, rendering, and painting. Recently, we've upgraded the Timeline with more instrumentation so that you can see a more in-depth view of your app's performance.

We've added the following features:

- integrated JavaScript profiler. (Flame chart included!)
- frame viewer to help you visualize composited layers.
- paint profiler for detailed drill-downs into the browser's painting activity.


Note that using the **Paint** capture options described in this article do incur some performance overhead, so flip them on only when you want 'em.

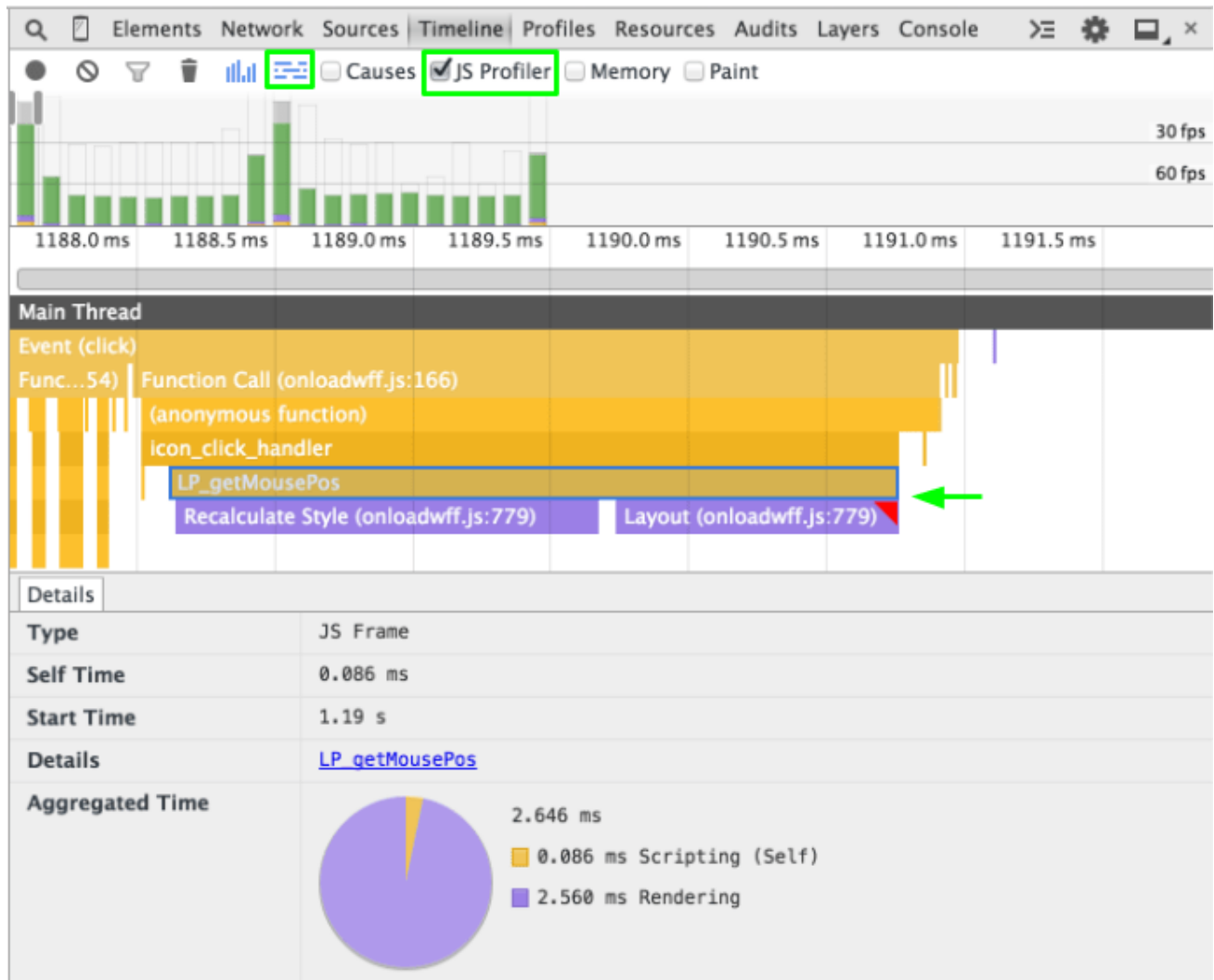
Integrated JavaScript Profiler

If you've ever poked around in **Profiles** panel, you're probably familiar with the JavaScript CPU profiler. This tool measures where execution time is spent in your JavaScript functions. By viewing JavaScript profiles with the Flame Chart, you can visualize your JavaScript processing over time.

Now, you can get this granular breakdown of your JavaScript execution in the **Timeline** panel. By selecting the **JS Profiler** capture option, you can see your JavaScript call stacks in the Timeline along with other browser events. Adding this feature to the Timeline helps streamline your debugging workflow. But more than that, it allows you to view your JavaScript in context and identify the parts of your code that affect page load time and rendering.

In addition to the JavaScript profiler, we also integrated a Flame Chart view into the **Timeline** panel. You can now view your app's activity either as the classic waterfall of


events or as a Flame Chart. The Flame Chart icon  allows you to toggle between these two views.



Using the **JS Profiler** capture option and Flame Chart view to investigate call stacks in the Timeline.

Note: Use WASD to zoom and pan through the Flame Chart. Shift-drag to draw a selection box.

Frame Viewer

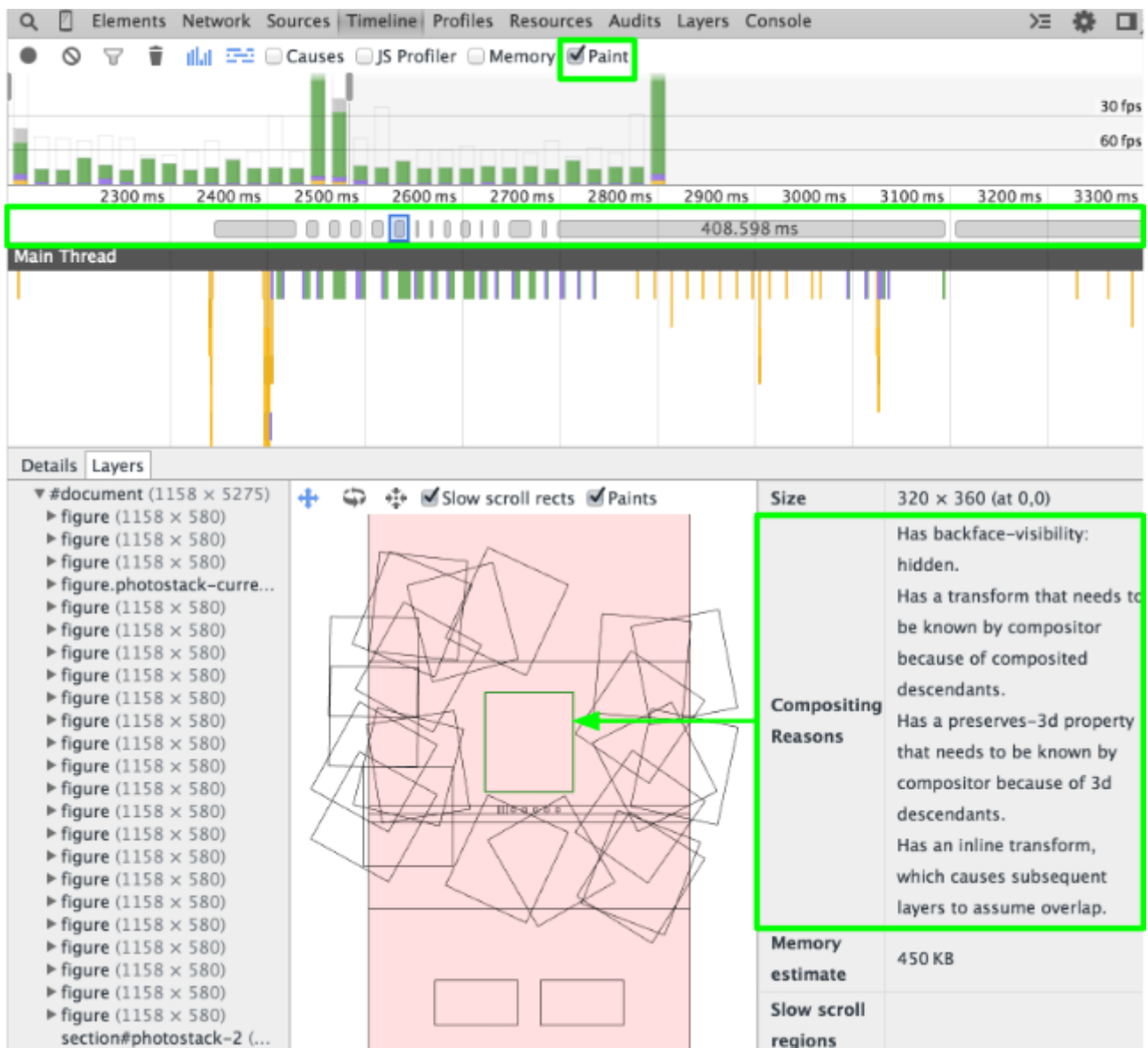
The art of [layer compositing](#)  is another aspect of the browser that has been mostly hidden from developers. When used sparingly and with care, layers can help avoid costly repaints and yield huge performance boosts. But it's often not obvious to predict how the

browser will composite your content. Using the Timeline's new **Paint** capture option, you can visualize composited layers at each frame of a recording.

When you select a gray frame bar above the **Main Thread**, its **Layers** panel provides a visual model of the layers that compose your app.

Note: Play back animations by clicking through frame bars on a Timeline recording.

You can zoom, rotate, and drag the layers model to explore its contents. Hovering over a layer reveals its current position on the page. Right-clicking on a layer lets you jump to the corresponding node in the **Elements** panel. These features show you what was promoted to a layer. If you select a layer, you can also see why it was promoted in the row labeled **Compositing Reasons**.

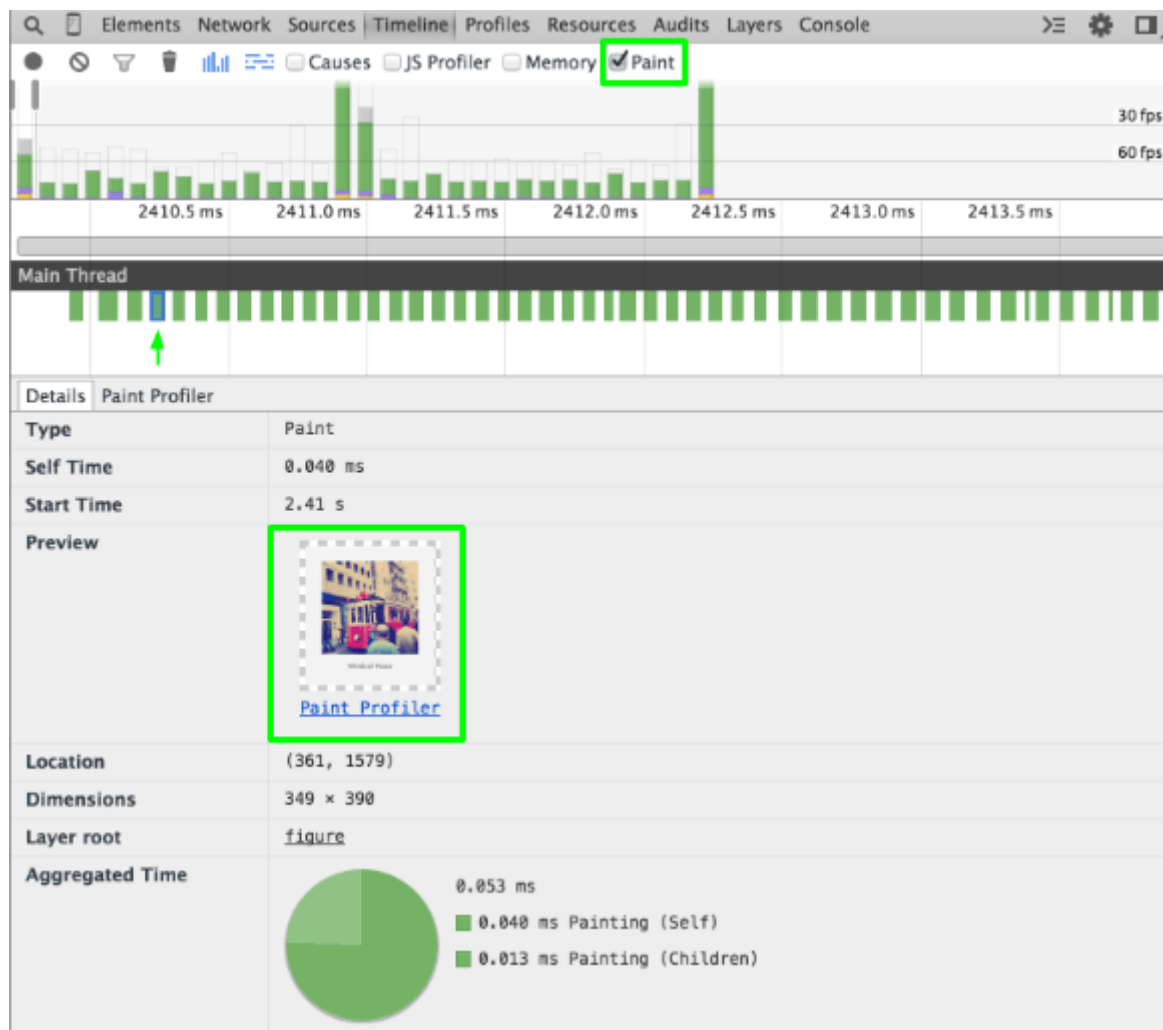


Inspecting a layer from [Codrops' Scattered Polaroids Gallery](#) to reveal the browser's reasons for compositing.

Paint Profiler

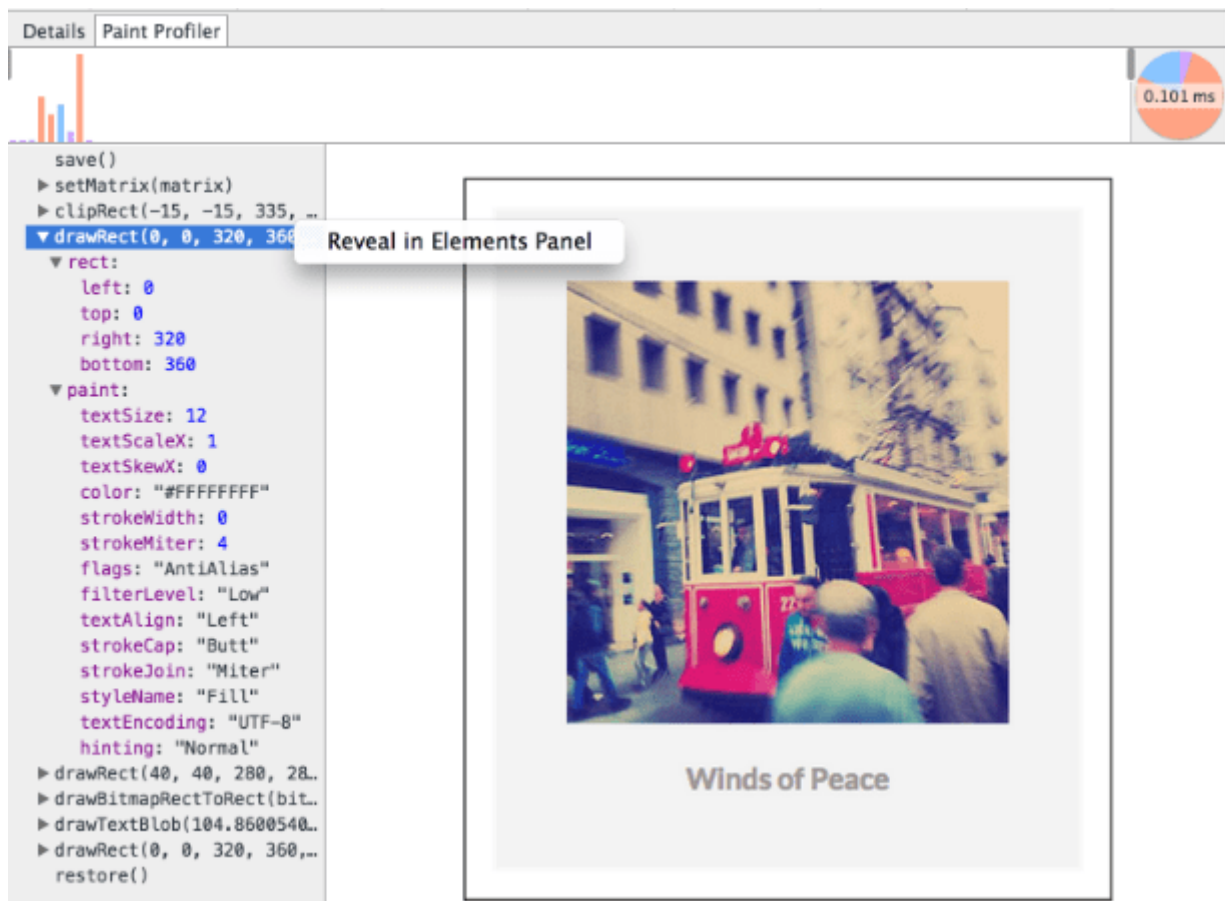
Last but not least, we've added the paint profiler to help you identify jank caused by expensive paints. This feature enriches the Timeline with more details about the work Chrome does during paint events.

For starters, it's now easier to identify the visual content corresponding to each paint event. When you select a green paint event in the Timeline, the **Details** pane shows you a preview of the actual pixels that were painted.



Previewing pixels that the browser painted using the **Paint** capture option.

If you really want to dive in, switch over to the **Paint Profiler** pane. This profiler shows you the exact draw commands that the browser executed for the selected paint. To help you connect these native commands with actual content in your app, you can right-click on a **draw*** call and jump straight to the corresponding node in the **Elements** panel.



Relating native browser **draw*** calls to DOM elements using the **Paint Profiler**.

The mini-timeline across the top of the pane lets you play back the painting process and get a sense of which operations are expensive for the browser to perform. Drawing operations are color-coded as follows: **pink** (shapes), **blue** (bitmap), **green** (text), **purple** (misc.). Bar height indicates call duration, so investigating tall bars can help you understand what about a particular paint was costly.

Profile and profit!

When it comes to performance optimization, knowledge of the browser can be incredibly powerful. By giving you a peek under the hood, these Timeline updates help clarify the relationship between your code and Chrome's rendering processes. Try out these new options in the Timeline and see what Chrome DevTools can do to enhance your jank-hunting workflow!

Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](#), and code samples are licensed under the [Apache 2.0 License](#). For details, see our [Site Policies](#). Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 2, 2018.