

Complexities of an Infinite Scroller



By Surma

Surma is a contributor to WebFundamentals



By Robert Flack

Robert is a contributor to WebFundamentals

TL;DR: Re-use your DOM elements and remove the ones that are far away from the viewport. Use placeholders to account for delayed data. Here's a [demo](#) and the [code](#) for the infinite scroller.

Infinite scrollers pop up all over the internet. Google Music's artist list is one, Facebook's timeline is one and Twitter's live feed is one as well. You scroll down and before you reach the bottom, new content magically appears seemingly out of nowhere. It's a seamless experience for users and it's easy to see the appeal.

The technical challenge behind an infinite scroller, however, is harder than it seems. The range of problems you encounter when you want to do The Right Thing™ is vast. It starts with simple things like the links in the footer becoming practically unreachable because content keeps pushing the footer away. But the problems get harder. How do you handle a resize event when someone turns their phone from portrait to landscape or how do you prevent your phone from grinding to a painful halt when the list gets too long?

The right thing™

We thought that was reason enough to come up with a reference implementation that shows a way to tackle all these problems in a reusable way while maintaining performance

standards.

We are going to use 3 techniques to achieve our goal: DOM recycling, tombstones and scroll anchoring.

Our demo case is going to be a Hangouts-like chat window where we can scroll through the messages. The first thing we need is an infinite source of chat messages. Technically, none of the infinite scrollers out there are *truly* infinite, but with the amount of data that is available to get pumped into these scrollers they might as well be. For simplicity's sake we will just hard-code a set of chat messages and pick message, author and occasional image attachment at random with a sprinkle of artificial delay to behave a little bit more like the real network.


978 DOM Nodes (5)

Why not

Thu Jun 09 2016 18:49:44 GMT+0100 (BST)


And that we'll do so

Thu Jun 09 2016 18:50:13 GMT+0100 (BST)




The defer mean to your Custom Elements JavaScript says we don't have

Thu Jun 09 2016 18:50:32 GMT+0100 (BST)




Where is being run



Thu Jun 09 2016 18:50:45 GMT+0100 (BST)

At least trying new Promise

Thu Jun 09 2016 18:51:08 GMT+0100 (BST)



in the attached

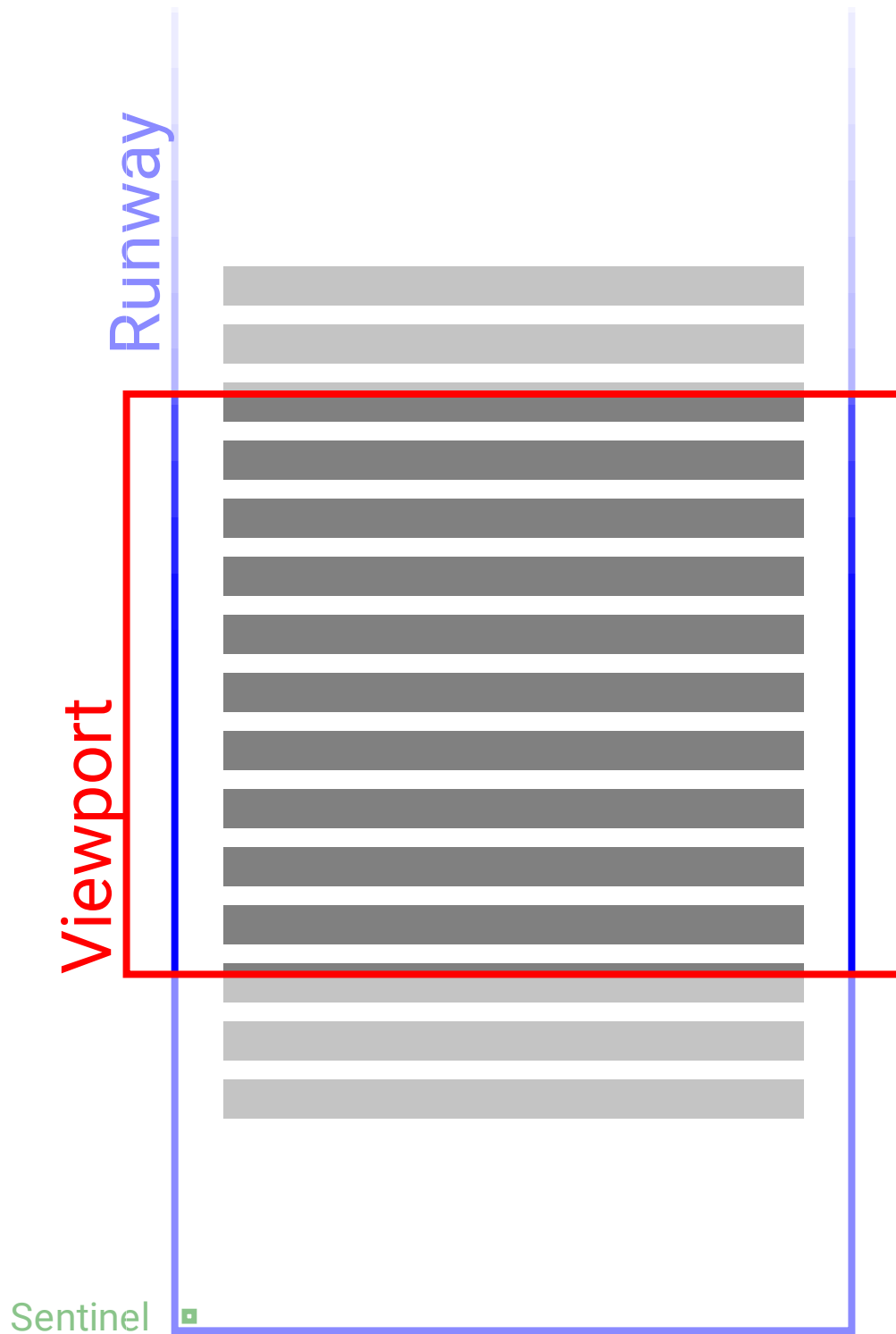
Thu Jun 09 2016 18:51:26 GMT+0100 (BST)

DOM recycling

DOM recycling is a underutilized technique to keep the DOM node count low. The general idea is to use already created DOM elements that are off-screen instead of creating new ones. Admittedly, DOM nodes themselves are cheap, but they are not free, as each of them adds extra cost in memory, layout, style and paint. Low-end devices will get noticeably slower if not completely unusable if the website has too big of a DOM to manage. Also keep in mind that every relayout and reapplication of your styles – a process that is triggered whenever a class is added or removed from a node – grows more expensive with a bigger DOM. Recycling your DOM nodes means that we are going to keep the total number of DOM nodes considerably lower, making all these processes faster.

The first hurdle is the scrolling itself. Since we will only have a tiny subset of all available items in the DOM at any given time, we need to find another way to make the browser's scrollbar properly reflect the amount content that is theoretically there. We will use a 1px by 1px sentinel element with a transform to force the element that contains the items – the runway – to have the desired height. We will promote every element in the runway to their own layer to make sure the layer of the runway itself completely empty. No background color, nothing. If the runway's layer is non-empty it is not eligible for the browser's optimizations and we will have to store a texture on our graphics card that has a height of a couple of hundred thousand pixels. Definitely not viable on a mobile device.

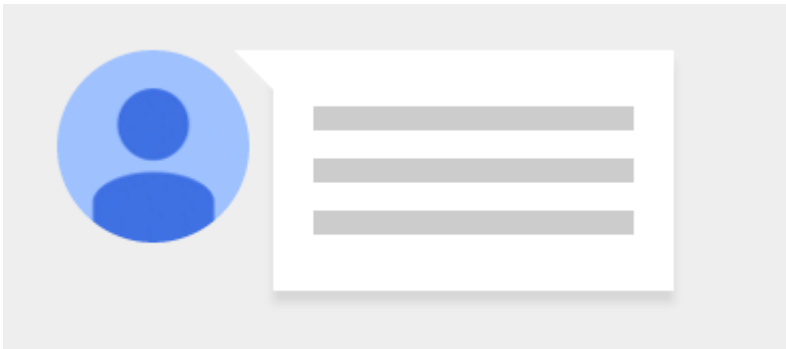
Whenever we scroll, we will check if the viewport has come sufficiently close to the end of the runway. If so, we will extend the runway by moving the sentinel element and moving the items that have left the viewport to the bottom of the runway and populate them with new content.



The same goes for scrolling in the other direction. We will, however, never shrink the runway in our implementation, so that the scrollbar position stays consistent.

Tombstones

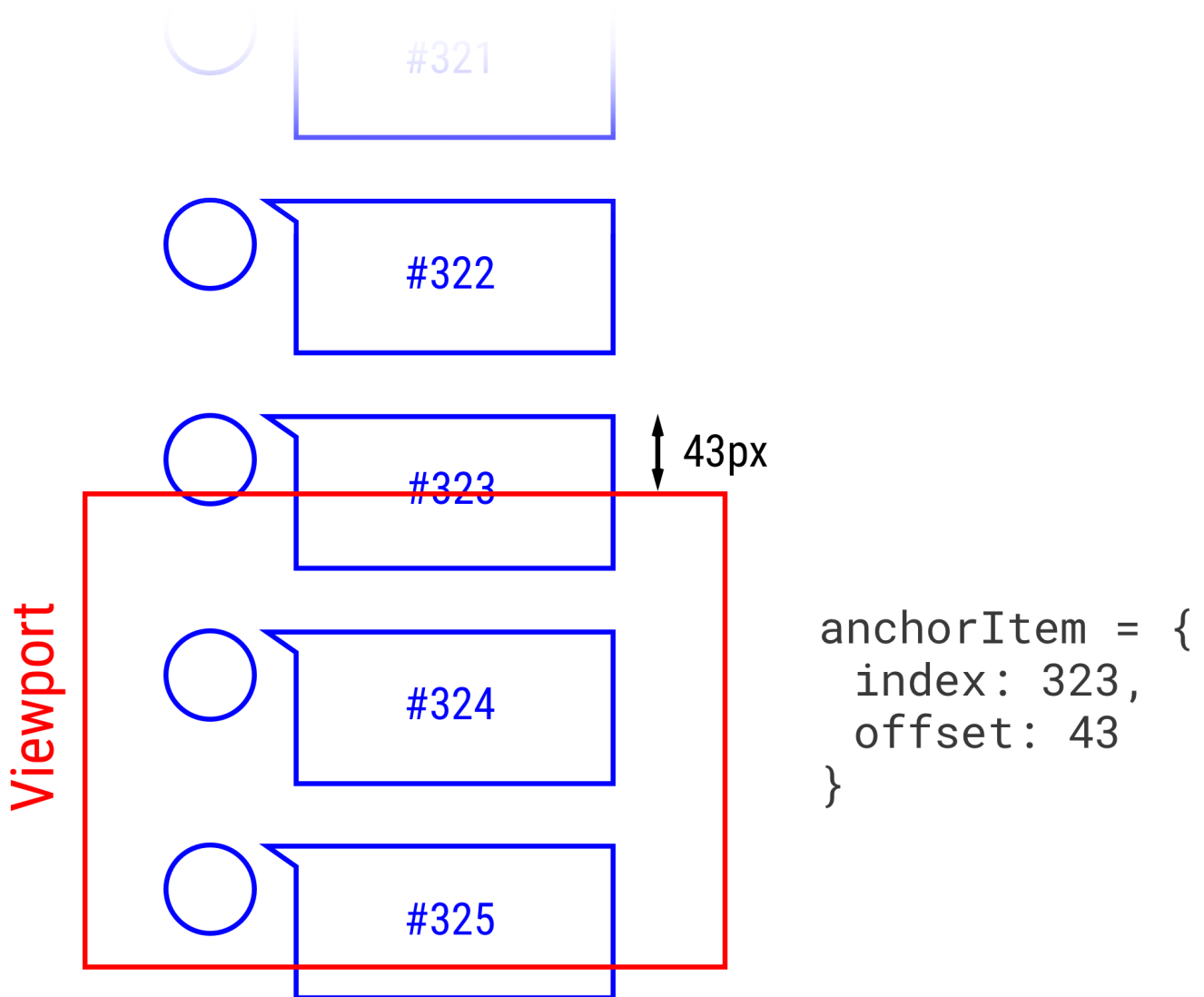
As we mentioned earlier, we try to make our data source behave like something in the real world. With network latency and everything. That means that if our users make use of flicky scrolling, they can easily scroll past the last element we have data for. If that happens, we will place a tombstone item – a placeholder – that will get replaced by the item with actual content once the data has arrived. Tombstones are also recycled and have a separate pool for re-usable DOM elements. We need that so we can make a nice transition from a tombstone to the item populated with content, which would otherwise be very jarring to the user and might actually make them lose track of what they were focusing on.



An interesting challenge here is that real items can have a bigger height than the tombstone item because of differing amounts of text per item or an attached image. To resolve this, we will adjust the current scroll position every time data comes in and a tombstone is being replaced above the viewport, *anchoring* the scroll position to an element rather than a pixel value. This concept is called scroll anchoring.

Scroll anchoring

Our scroll anchoring will be invoked both when tombstones are being replaced as well as when the window gets resized (which also happens when the device is being flipped!). We will have to figure out what the top-most visible element in the viewport is. As that element could only be partially visible, we will also store the offset from the top of the element where the viewport begins.



If the viewport is resized and the runway has changes, we are able to restore a situation that feels visually identical to the user. Win! Except a resized window means that each item has potentially changed its height, so how do we know how far down the anchored content should be placed? We don't! To find out we would have to layout every element above the anchored item and add up all of their heights; this could cause a significant pause after a resize, and we don't want that. Instead, we resort to assuming that every item above is the same size as a tombstone and adjust our scroll position accordingly. As elements are scrolled into the runway, we adjust our scroll position, effectively deferring the layout work to when it is actually needed.

Layout

I have skipped over an important detail: Layout. Each recycling of a DOM element would normally relayout the entire runway which would bring us well below our target of 60 frames

per second. To avoid this, we are taking the burden of layout onto ourselves and use absolutely positioned elements with transforms. This way we can pretend that all the elements further up the runway are still taking up space when in actuality there is only empty space. Since we are doing layout ourselves, we can cache the positions where each item ends up and we can immediately load the correct element from cache when the user scrolls backwards.

Ideally, items would only get repainted once when they get attached to the DOM and be unfazed by additions or removals of other items in the runway. That is possible, but only with modern browsers.

Bleeding-edge tweaks

Recently, [Chrome added support for CSS Containment](#), a feature that allows us developers to tell the browser an element is a boundary for layout and paint work. Since we are doing layout ourselves here, it's a prime application for containment. Whenever we add an element to the runway, we *know* the other items don't need to be affected by the relayout. So each item should be get `contain: layout`. We also don't want to affect the rest of our website, so the runway itself should get this style directive as well.

Another thing we considered is using [IntersectionObservers](#) as a mechanism to detect when the user has scrolled far enough for us to start recycling elements and load new data. However, IntersectionObservers are specified to be high latency (as if using `requestIdleCallback`), so we might actually *feel* less responsive with IntersectionObservers than without. Even our current implementation using the `scroll` event suffers from this problem, as scroll events are dispatched on a "best effort" basis. Eventually, [Houdini's Compositor Worklet](#) would be the high fidelity solution to this problem.

It's still not perfect

Our current implementation of DOM recycling is not ideal as it adds all elements that *pass* through the viewport, instead of just caring about the ones that are actually *on* screen. This means, that when you scroll *reaaaally* fast, you put so much work for layout and paint on Chrome that it can't keep up. You will end up seeing nothing but the background. It's not the end of the world but definitely something to improve on.

We hope you see how challenging simple problems can become when you want to combine a great user experience with high performance standards. With Progressive Web Apps becoming first-class citizens on mobile phones, this will become more important and web developers will have to continue investing into using patterns that respect performance constraints.

All the code can be found in our [repository](#). We have done our best to keep it reusable, but won't be publishing it as an actual library on npm or as a separate repo. The primary use is educational.

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.