# Tips and tricks for improving JavaScript performance

## Points on startup performance

**Combining Cascading Style Sheets (CSS)** helps reducing loading time, especially if a large number of UI controls are used.

**Code combination** reduced the load time of 50 files by about 25 per cent, in some example applications. Your number will probably be different — it may even be better.

**Code minification** benefits will depend on the code. If you get some tens of per cent here, another 25 per cent for code combination, and so on, you are already well on the way to improvement!

**Lazy loading** is pretty much a must-have to guarantee good startup. If you have 1 MB of code you will spend about 10 seconds just loading that code. However, you probably don't need more than 100 K, at most, to show the initial view — that's 1 second's worth of loading time. Lazy loading lets you defer and scatter the 9 seconds.

**Distinguish between 'startup' and 'right after startup'**, and move as much as possible to 'right after startup'. By designing the UI so that it can be updated when data changes (model change events), you can usually slap up the UI quite a bit sooner than if you first have to wait for everything to be ready and only then show the UI. Usually there are very few things you can't do either lazily or 'right after startup'. The important thing is the perceived startup time, not the actual startup time.

Think about it this way: A simple Hello World widget an start in 2.5 seconds. Rethink your application so that it starts out as a Hello World that shows something that looks like the real UI. Then, over the next few seconds, load enough logic and data that you can fill in the blanks in the UI and make it look fully loaded. This data can be a quick-cached version, and the logic that puts it into the UI doesn't have to be extravagant. The key is getting something onto screen fast.

While trying to get the splash up as soon as technically possible, we tried using a trick from an example application — that is, starting the application as if it was a plain Hello World with no code whatsoever and then injecting the actual loading code some milliseconds after that 'minimal' widget has been loaded. We found, to our surprise, that the couple-millisecond delay actually had a very interesting side effect. With certain delay values, the entire application would start up two seconds faster, and in some cases the splash would appear two seconds faster. It is worth mentioning that even with small changes this appears to change behaviour. For example, setting the delay using an onload event removes the effect. Instead doing it at the bottom of <body..</body works.

A final key point regarding startup is to avoid using the network at all cost. For the example application, the startup time is roughly as follows:

- Three seconds from the app grid and initial splash being loaded.
- Five seconds to load and evaluate the JavaScript™ code.

At this point, the application throws up a UI that is a mock of the actual main view; the user feels like the loading is 'almost done'.

- Two seconds, ideally, to do a handshake with the server, establish a session, and get the content for the main view (this includes an 'Are there updates for my client?' check).

Under real-life conditions, however, starting the internet connection, connecting to the server, getting the response back, and so on, may take as long as 10 seconds. So for most users, it takes 20 seconds to load, not 10. And not the 8 seconds it would take if there was no network connection necessary before showing the main view. This illustrates why it is so crucial to be able to display the UI **before** you need to use the network.

## Points on general performance

Basically there are three scrolling modes:

1. Full redraw
2. Copy scroll
3. Tile scroll

Full redraw means that the entire portion of the document that is in the viewport gets rendered for each frame when you scroll. If it takes 100 milliseconds to render, then you get 10 frames per second.

Copy scroll means that the previous rendered frame is copied slightly offset from where it was in the previous frame (offset by the


(C) Copyright Nokia 2014. All rights reserved.

amount you just scrolled), and then the new area of the document that has come into the view is rendered 'from scratch'. In theory, if 10 per cent of the view is 'new' and 90 percent was visible already in the previous frame, then you just need to render 10 per cent in addition to copying the old frame. In theory, if it takes 100 ms to render the full view, then it takes 10 ms to render 10 per cent of it. In real life, of course, it is not quite this simple, but you get the basic idea. In short, the copy scroll method is faster than full redraw.

Tile scroll is like copy scrolling on a higher level, except that instead of a single 'tile' (the old frame), you have multiple smaller ones that not only cover the viewport but also some margin above and below the viewport. Most of the time when you scroll it is sufficient to just move the tiles around. Only when you've scrolled very far do you need to render more of the document to the tiles. This is the fastest option, and the Nokia N900 device, for example, uses this mode. When empty areas with that checkerboard pattern scroll into view, you know that you've run out of tiles.

Unfortunately tile scrolling in not yet supported in Symbian devices, so the best option is to get copy scrolling. The problem is that your content may be such that copy scrolling isn't an option. This can happen, for example, when you place a fixed element in the document, or a fixed background, or in fact anything that makes it so that just moving the previously rendered frame won't be sufficient to produce a portion of the next frame. By the way, the same things that ruin copy scrolling also ruin tile scrolling.

If this happens, fall back to full redraw mode.

Understanding the scrolling modes and knowing which ones are available and in what contexts on a particular device is crucial to knowing how to arrange your content to get the best rendering performance.

For example, consider using IFrames for the scrollable content if that gives you copy or tile scrolling for the scrollable part of the document. If you get nothing but full redraws, then it is probably simpler to arrange the UI in a single document. That way you can avoid the overhead of creation and destruction of IFrames, loading CSS twice, dealing with cross-window and cross-document issues, and so on.

## Considerations for DOM

**Creation and destruction of DOM** is quite slow, so UI elements should take this into account and be designed so they do not create unnecessary DOM. Basically, use lazy functionality here, too, so that if a snippet can have various states, those states (or at least the less common states) only get created when needed. This also reduces memory overhead and destruction time.

While it is obviously important that creation happens immediately, the same is not necessarily true for destruction. In principle, there is no reason why destruction can't be delayed until after creation of the new view and switching to it. In many cases, DOM destruction is actually slower than DOM creation, which means that by deferring destruction until after the view switch you can sometimes cut view switch times in half. Some applications use this technique for recycling.

And while we are on the topic of recycling....

For items that are used a lot, such as content item snippets, consider **recycling**. It is quite a bit faster to simply recycle an existing piece of DOM than to create a new subtree and then destroy it later.

Some people create very complex snippets that use a lot of bindings. While this might make the snippet simple to use and the code short, there's a small performance penalty for each binding. So try not to go overboard with bindings, especially for things that are used rarely (such as some convenience binding to styles or similar). Frequently, a simple data-bind-ref to a node will be enough, and data-bind-ref is one of the fastest binding types anyway. The same applies for custom accessors — avoid them unless necessary.

**Consider not creating any DOM that is outside the viewport.** There are many ways to accomplish this. For example, you could create an initial set of DOM prior to the view switch that covers only the viewport. Immediately after, create more DOM, and for the portion outside the viewport. If you have, for example, 50 items in the view and 10 fit inside the viewport, then this will make your view switch time five times faster — for example, 100 ms for the top 10 items — and then create the next 40 items 10 at a time over the next two seconds. You could combine this with pooling and, for example, always have 10 items ready in a pool and balance memory and DOM creation time.

Another example of how to create DOM dynamically is to use **endless scrolling**. Have a model reflect the full view data but only create a viewport-full or two. Then, as the user scrolls down and hits the bottom, add more DOM to the bottom. But again, only a viewport-full or two. That way you always create only a bit of DOM at a time.

Our example application uses endless scrolling to scroll two types of items. One type might have hundreds or thousands of items in it, and creating all of them is not only too slow but also uses too much memory. Instead, only a handful of items (the amount that fits in the viewport plus some margin) are created and then moved around and updated as the user scrolls the list. This gives the impression that there are hundreds or thousands of items, while in fact there may be only 25 or so.

This list is implemented so that there is a container div that has its height set to the total height of all items in the data model.


(C) Copyright Nokia 2014. All rights reserved.

Within this list is a set of ‘renderer’ snippets that are moved inside the container div using absolute positioning based on the current viewport position. For example, if the viewport is at scrollY 0 and the viewport is 500 pixels in height and an item is 50 pixels in height, then we move 10 of the renderer snippets to positions top=0, top=50, etc., and update their data to reflect items 0, 1, 2 ... from the data model. If you scroll to position 500, the items would be 10, 11, 12, etc., and the absolute positioning would be 500, 550, 600, and so on.

The way the list works is similar to tile scrolling, and it has some margin (one viewport above and one below) so that the repositioning and data updates don't need to be done too often. The result is a list component that looks and behaves exactly as if it had thousands of items in it, despite having only a handful. Because the container div is sized to the full list length, scrollbars and kinetic scrolling will behave just as if the list had all of the items in it.

A number of algorithmic issues in the example application also resulted in pretty decent speedups. For example, sorting a list used to take 40 seconds, but after optimisation took less than 40 milliseconds — a 1000x time speedup. Key to these improvements: **Don't call functions unless you have to, don't repeat work, and don’t create unnecessary objects that exercise the garbage collector.** Previously, 1500 items took 20 seconds to sort; after optimisation there was again a roughly 1000x time speedup. The key here was the sorting but also not using regular expressions.

Once in a while you have code that is 100 or even 1000 times slower than an ideal implementation. With C/C++ that might mean you're at 100 ms instead of 0.1 ms. With JavaScript you might start out at 10 ms and being off by 1000 means you're at 10 seconds. The 10-second delay is very obvious, and it is easy to blame JavaScript or the platform (especially if the platform is known to be slow in other areas), whereas the real issue is that the code is very suboptimal. In short, with JavaScript it becomes even more important than normal to make sure algorithms are smart.

Finally, and most importantly:

**Network latency** kills performance. If you have to do multiple requests, then do them on the server and aggregate them together to a single request between client and server.

Back to JavaScript Performance Best Practices


(C) Copyright Nokia 2014. All rights reserved.