Adding Interactivity with JavaScript

By Ilya Grigorik

Ilya is a Developer Advocate and Web Perf Guru

JavaScript allows us to modify just about every aspect of the page: content, styling, and its response to user interaction. However, JavaScript can also block DOM construction and delay when the page is rendered. To deliver optimal performance, make your JavaScript async and eliminate any unnecessary JavaScript from the critical rendering path.

TL;DR

- JavaScript can query and modify the DOM and the CSSOM.
- JavaScript execution blocks on the CSSOM.
- JavaScript blocks DOM construction unless explicitly declared as async.

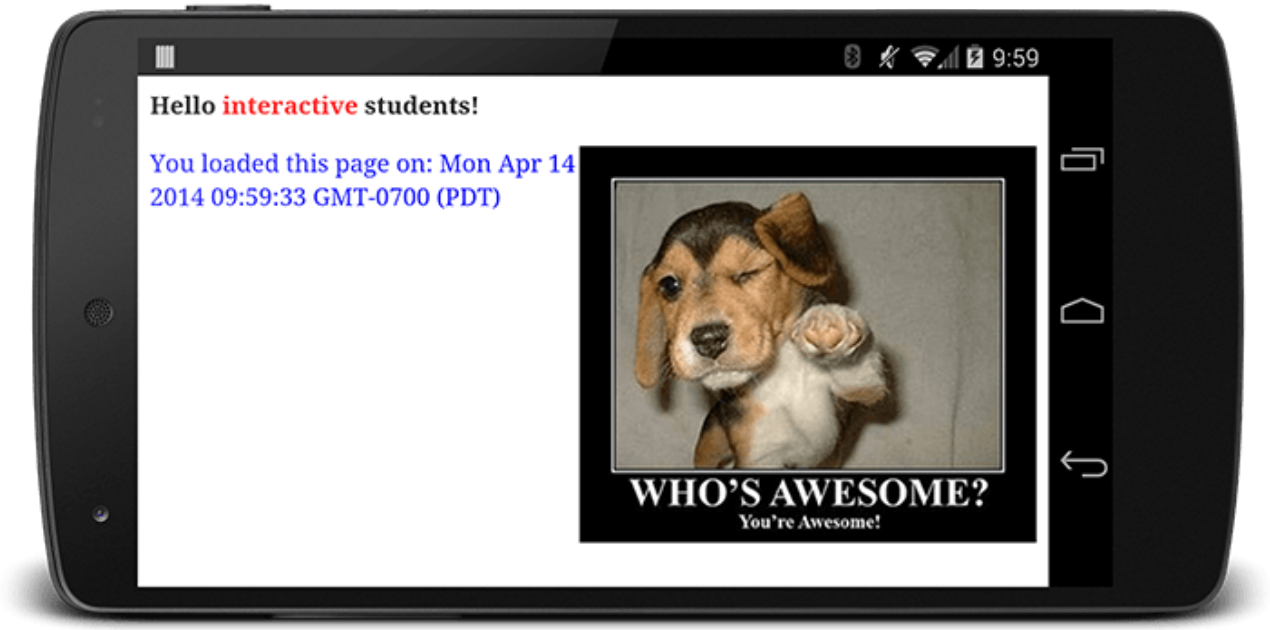
JavaScript is a dynamic language that runs in a browser and allows us to alter just about every aspect of how the page behaves: we can modify content by adding and removing elements from the DOM tree; we can modify the CSSOM properties of each element; we can handle user input; and much more. To illustrate this, let's augment our previous "Hello World" example with a simple inline script:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path: Script</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script>
      var span = document.getElementsByTagName('span')[0];
      span.textContent = 'interactive'; // change DOM text content
      span.style.display = 'inline';  // change CSSOM property
      // create a new element, style it, and append it to the DOM
      var loadTime = document.createElement('div');
      loadTime.textContent = 'You loaded this page on: ' + new Date();
      loadTime.style.color = 'blue';
```

```
      document.body.appendChild(loadTime);
    </script>
  </body>
</html>
```

Try it

- JavaScript allows us to reach into the DOM and pull out the reference to the hidden span node; the node may not be visible in the render tree, but it's still there in the DOM. Then, when we have the reference, we can change its text (via .textContent), and even override its calculated display style property from "none" to "inline." Now our page displays "**Hello interactive students!**".
- JavaScript also allows us to create, style, append, and remove new elements in the DOM. Technically, our entire page could be just one big JavaScript file that creates and styles the elements one by one. Although that would work, in practice using HTML and CSS is much easier. In the second part of our JavaScript function we create a new div element, set its text content, style it, and append it to the body.



With that, we've modified the content and the CSS style of an existing DOM node, and added an entirely new node to the document. Our page won't win any design awards, but it illustrates the power and flexibility that JavaScript affords us.

However, while JavaScript affords us lots of power, it creates lots of additional limitations on how and when the page is rendered.

First, notice that in the above example our inline script is near the bottom of the page. Why? Well, you should try it yourself, but if we move the script above the *span* element, you'll

notice that the script fails and complains that it cannot find a reference to any *span* elements in the document; that is, *getElementsByTagName('span')* returns *null*. This demonstrates an important property: our script is executed at the exact point where it is inserted in the document. When the HTML parser encounters a script tag, it pauses its process of constructing the DOM and yields control to the JavaScript engine; after the JavaScript engine finishes running, the browser then picks up where it left off and resumes DOM construction.

In other words, our script block can't find any elements later in the page because they haven't been processed yet! Or, put slightly differently: **executing our inline script blocks DOM construction, which also delays the initial render.**

Another subtle property of introducing scripts into our page is that they can read and modify not just the DOM, but also the CSSOM properties. In fact, that's exactly what we're doing in our example when we change the display property of the span element from none to inline. The end result? We now have a race condition.

What if the browser hasn't finished downloading and building the CSSOM when we want to run our script? The answer is simple and not very good for performance: **the browser delays script execution and DOM construction until it has finished downloading and constructing the CSSOM.**

In short, JavaScript introduces a lot of new dependencies between the DOM, the CSSOM, and JavaScript execution. This can cause the browser significant delays in processing and rendering the page on the screen:

- The location of the script in the document is significant.
- When the browser encounters a script tag, DOM construction pauses until the script finishes executing.
- JavaScript can query and modify the DOM and the CSSOM.
- JavaScript execution pauses until the CSSOM is ready.

To a large degree, "optimizing the critical rendering path" refers to understanding and optimizing the dependency graph between HTML, CSS, and JavaScript.

Parser blocking versus asynchronous JavaScript

By default, JavaScript execution is "parser blocking": when the browser encounters a script in the document it must pause DOM construction, hand over control to the JavaScript runtime, and let the script execute before proceeding with DOM construction. We saw this in

action with an inline script in our earlier example. In fact, inline scripts are always parser blocking unless you write additional code to defer their execution.

What about scripts included via a script tag? Let's take our previous example and extract the code into a separate file:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path: Script External</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script src="app.js"></script>
  </body>
</html>
```

app.js

```
var span = document.getElementsByTagName('span')[0];
span.textContent = 'interactive'; // change DOM text content
span.style.display = 'inline';  // change CSSOM property
// create a new element, style it, and append it to the DOM
var loadTime = document.createElement('div');
loadTime.textContent = 'You loaded this page on: ' + new Date();
loadTime.style.color = 'blue';
document.body.appendChild(loadTime);
```

Try it

Whether we use a <script> tag or an inline JavaScript snippet, you'd expect both to behave the same way. In both cases, the browser pauses and executes the script before it can process the remainder of the document. However, **in the case of an external JavaScript file the browser must pause to wait for the script to be fetched from disk, cache, or a remote server, which can add tens to thousands of milliseconds of delay to the critical rendering path.**

By default all JavaScript is parser blocking. Because the browser does not know what the script is planning to do on the page, it assumes the worst case scenario and blocks the parser. A signal to the browser that the script does not need to be executed at the exact point where it's referenced allows the browser to continue to construct the DOM and let the script execute when it is ready; for example, after the file is fetched from cache or a remote server.

To achieve this, we mark our script as *async*:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path: Script Async</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script src="app.js" async></script>
  </body>
</html>
```

Try it

Adding the async keyword to the script tag tells the browser not to block DOM construction while it waits for the script to become available, which can significantly improve performance.

Previous
← Render-Blocking CSS

Next
Measuring the Critical Rendering Path →

Last updated July 2, 2018.