WatchMeCode's **3 Rules for Using New JavaScript Features**

JavaScript is rapidly evolving. And with the TC39 Working Group setting the course for the language, and the larger community being involved in the process, it can be overwhelming at times.

Browsers, Node.js and other JavaScript runtime environments do their best to keep up with new features, but that doesn't mean all features are readily available to all users.

So, how do you know when a new JavaScript feature is ready for production?

Is it safe to use an early stage feature? Or should you wait until that feature is readily available in all browsers, across nearly all of your user base?

Unfortunately, the answer is not always simple. I do, however, have a few guidelines that I follow when evaluating a change to JavaScript… 3 rules that let me know when I should start using it.

But before I tell you those rules, it's important to understand the stages of a JavaScript feature.

JavaScript Feature Stages

If you're not familiar with it, the TC39 JavaScript Working Group has various stages for a language proposal / change.

- **Stage 0** is basically an idea
- **Stage 1** describes the problem and solution in broad strokes with some specification
- **Stage 2** provides a solid specification for the change using formal language
- **Stage 3** is where implementation ramps up and feedback from people using it is required
- **Stage 4** means the feature is done and ready to be officially part of the ECMAScript standard

With an understanding of the stages, you can better evaluate which features are safe and unsafe for production, starting with rule 1 and early stage features.

Rule #1: Start with Stage 3 and 4

The first rule is that I never (ok, rarely) use anything from Stage 2 or below.

There's just too much risk involved in these features and changes. It's highly likely that implementations will change and things that I'm doing now will be broken later.

That doesn't mean early-stage features aren't worth looking at or learning if there is some implementation of them, somewhere. But it does mean that I would avoid it for production use.

Beyond the simple rule of Stage 3 and 4 only, things get more complicated.

Rule #2: Use Kangax' Compatibility Table

My second rule - a shortcut really - to knowing whether I can use a feature is to first look at one of two websites:

- **Kangax' Compatibility Table** for all things JavaScript compatibility
- **Node.green** which is based on Kangax' table, but specific to Node.js

In these two sites - which I admit are difficult to read if you're not used to them - you'll find a cross-section of features and compatibility based on runtime environment and version.

It's a fair bet to say that a feature showing up as green (usable) in these lists is going to be stage 3 or 4. Some stage 2 features might show up as green, but these are likely going to be "simple" features like string padding.

A cross-section like this isn't always the best way to go about making your decision, though. I also have one more rule when evaluating features for use in my projects.

Rule #3: Syntax Sugar, Not Significant New Behavior

The last rule I use to determine whether I should be using a new feature is to judge whether this new feature is just syntax sugar or if it truly is new behavior, requiring significantly new code in a JavaScript runtime.

This is where the gray area and "it depends" really hits hard and makes it difficult to judge.

For example, string padding is just syntax sugar. Sure, it's technically a new method on an object, but that's such a simple behavior to add that any polyfill or precompiler worth anything will add it very quickly.

The same goes or ES6 classes, arrow functions, rest / spread params and even destructuring assignment. These are ultimately just syntax sugar.

But generators, on the other hand? There is significant new behavior in these… behavior that does more than just change the syntax. It creates a fundamental shift in how code can be executed - both synchronously and asynchronously.

I avoided generators until they were stable in Node.js because the transpilers and polyfill libraries added significant overhead in the form of 3rd party libraries, to use them.

Should You Avoid The "Unstable" Features?

The question of whether you should avoid a feature entirely is difficult to answer.

I would generally say to avoid early stage (0, 1 and most of 2) for production use. But even stage 2 is sometimes useful. And stage 3 and 4 are not always ready for prime time use, because of adoption rates in browsers and Node.js for new technology.

That doesn't mean you shouldn't learn these features and syntax options, though.

There are safe ways to learn these experimental and unstable changes - primarily through virtualization, whether it's a full virtual machine or through a virtualized application "container" such as with Docker.

Exploring options for working with new and experimental features is a topic for another day, however.

- Derick
- WatchMeCode