### https://angularjs.org/Changelog:

Feb 3:

* Sent doc out for review
* Re-worked the benchmark.js section

Feb 4: ongoing work:

* re-work ideas to address sampling benchmarks
* add section re webdriver tests
* Telemetry in “prior art”
* Clarify what a “macrobenchmark” is.
* Add a “We are not alone” section, listing partners.

Benchpress Design Doc

*Status: Draft II*

*Author: James deBoer <*[*deboer@google.com*](mailto:deboer@google.com)*, @*[*jbdeboer*](https://github.com/jbdeboer)*>*

# Objective

There are two goals for this project.

* ~~Build a web benchmarking system which produces actionable, meaningful, accessible results.~~
* Create a benchmark suite to ensure that Angular 2.0 apps are as performant as possible.

# Background

## Unit Testing For Performance

Unit testing enables correct systems. The first thing I learnt working on the Angular team is that unit testing is a key part of the development process. Over time, while iterating on architecture, we have built up a large suite of unit tests. These unit tests are critical to keeping the code malleable. Efforts, such as implementing Angular in Dart, would not have been possible without the unit test suite.

In addition to the large test suite, the Angular team has also invested heavily in Karma, our test runner. This allows us to run 2700 tests in 6 seconds. The speed and agility of Karma make it easy to include unit testing as an integral part of our development workflow.

Unit tests answer one question: “Does this code do what I expect?” Karma can answer that question quickly, across different browsers and different configurations. By constantly asking that question, our team is able to release correct systems.

## Benchmarking enables fast systems

As a parallel to unit testing, benchmarking enables fast systems. Benchmarking is a harder problem than unit testings; the tools and base of knowledge have not been built yet. However, there are enough parallels between benchmarking and unit testing that we can draw on unit testing knowledge for benchmarking.

As we’ve demonstrated with Karma on the Angular team, the key to a successful unit testing system is ease-of-use, speed and trustworthiness. Also, at a high level, a successful benchmarking system must answer one question, and answer that question well.

I believe that question is: “Is this code fast enough?”

## No More “Jank”: The Web at 60 Frames per Second

There is a push to make the web more interactive by eliminating “jank” to enable smooth animations and web performance. [jankfree.org](http://jankfree.org/) is a great resource cataloging current efforts.

In 2011, the “requestAnimationFrame” API was introduced in browsers, making it possible to tie animations and rendering to animation frames -- a concept from video games where programs must re-render the screen in between a display’s vertical sync to prevent screen tearing. On the web, this means Javascript programs are now aware of animation frames.

While dictated by hardware, most devices run at 60 frames per second. This means that applications have 1/60s = 16.6667ms between frames. If a Javascript routine plus screen layout plus repaint take longer than 16.6667ms, it will “push” the animation frame out, causing an animation to appear “janky”.

Since browsers won’t refresh any faster than 60 frames per second, being able to render at 60fps but not faster is the best goal. That means that any work a web app does must take at most 16.6667ms. Since we like whole numbers, we often refer to “16ms” instead.

# 

# Prior Art

We’ve been here before.

## benchmark.js

([link](http://benchmarkjs.com/))

The most popular web benchmarking library today is benchmark.js and its easy-to-use web frontend, jsperf.com.

benchmark.js allows developers to define a benchmark. The harness will run the benchmark multiple times to generate more robust data. The library enables micro-benchmarking by grouping runs into time-bound cycles which can be easily measured and applying tricks to de-optimize the VM.

There is a fair amount of opposition to benchmark.js and jsperf. While much of the ire is over microbenchmarks, benchmark.js/jsperf.com is often cited as it makes microbenchmarking easy.

* @mraleph [[1]](http://mrale.ph/blog/2013/08/14/hidden-classes-vs-jsperf.html) [[2]](http://mrale.ph/blog/2012/12/15/microbenchmarks-fairy-tale.html) [[3]](http://www.youtube.com/watch?v=65-RbBwZQdU)
* [reddit thread](http://my.reddit.com/r/javascript/comments/1npl7k/pitfalls_in_micro_benchmarking_or_why_jsperf_is/) with jdalton and mraleph

### Drawbacks

Benchmark.js will run benchmarks multiple times. This gives the VM more of a chance to optimize the code under test. Since production apps rarely run the same code in a tight loop, theses optimizations may lead to incorrect results.

## Octane / Kraken

([Kraken link](http://krakenbenchmark.mozilla.org/), [Octane link](https://developers.google.com/octane/))

Octane and Kraken are not web benchmarks. They are strictly Javascript benchmarks and most useful to VM developers.

However, Octane is a useful study in that it

* focuses on “jank” or 60fps apps, which is a goal of Benchpress
* Octane is maintained and used by the V8 team to measure optimizations.
* Kraken is maintained by Mozilla.

I believe that maintaining our own benchmark suite is key to a team’s performance goals. This is in contrast to third-party benchmark suites which don’t get as much attention from the developers that can effect the benchmarks.

## Robohornet

([link](http://www.robohornet.org/))

Robohornet is a inactive browser benchmark suite with many of the same goals as Benchpress. They had the laudable goal of improving browser performance through benchmark competition. However, they were pooh-poohed by the community mainly over the "microbenchmarks are lies" meme. [[1]](http://blogs.msdn.com/b/ie/archive/2011/02/08/focusing-on-real-world-web-performance-with-internet-explorer-9.aspx) [[2]](http://news.cnet.com/8301-1023_3-57519568-93/life-beyond-javascript-googles-abuzz-over-robohornet-test/)

## Browserscope

[(link)](http://www.browserscope.org/%20)

(also by Google) collects data in the wild -- doesn’t collect benchmarks, rather other browser stats.

## Learnings from Benchpress Draft I

The first attempt at Benchpress was a benchmark.js backed runner much like Robohornet. While I hoped that the benchmarks being run by the system were large enough to be significant, they still suffered from the same problems as microbenchmarks.

These benchmarks were loading entire apps and then testing a small part of the app by simulating user actions. However, in order for the framework (using benchmark.js) to compute meaningful statistics it needed to run the user action hundreds of times. During the run, the browser and Javascript engine were able to optimize the code and dramatically speed up the user action.

This gave the statistics a bimodal distribution were the first third (~100) runs would be ~3-4x slower than the last two-thirds of the runs (~200). Since most apps would not typically run 100 user actions, it was difficult interpret the results. The data showing the bimodal distribution was hidden inside the benchmark.js library, making it even more difficult to trust the numbers returned by the library.

## Performance in the Wild

The best source of performance data is from actual users using actual apps on actual devices. There are a number of systems for collecting and analyzing data from web app sessions.

There are a number of companies that help sites measure real-world performance

* Google Analytics
* New Relic
* Akamai
* ...

The huge benefit to these numbers is that they are representative of actual usage. However, there are a number of downsides:

* it is very hard to isolate performance changes. With a new app release, there may be hundreds of changes. Attributing performance changes to any one code change is difficult.
* it is impossible to measure changes across UI flow changes were a single code path may not exist from release to release.
* The feedback loop is very long since you need to push to production first and wait for data from the users.

Many of these downsides can be mitigated with a robust A/B experiment framework and continuous release cycle. However these require infrastructure and significant overhead in processes. These options are not available to framework developers (like Angular) who do not control individual apps.

# Detailed Design

## What Makes A Useful Benchmark?

### Actionable

Benchpress must produce more than a score or timing information. Instead it should be able to answer the binary question: “Is this code fast enough?”

This allows developers to take action -- make the code faster or move on to something else.

### Meaningful

Benchmarks should mimic real-world code as much as possible. They should run on real hardware using actual production code. When a benchmark improves (e.g. flips from being “not fast enough” to “fast enough”), that change should be reflected in real applications.

Benchmarks should also be trustworthy. Developers should be able to count on them to run the code they intended to benchmark. There must be some form of assertion library included in the framework.

They should also be reliable. Running a benchmark twice should give the same results.

### Accessible

Benchpress should make it easy for other developers to run the benchmarks and verify the results. This is especially important with respect to new hardware.

## Macro-benchmarks

The web performance community has worked very hard to debunk microbenchmarks. But at the same time, through jsperf.com and benchmarkjs, microbenchmarks are the most accessible data sources that developers have access to.

There is a volume of literature pointing out the flaws with microbenching, but very few examples on how to write a trustworthy, useful non-micro benchmark.

In order to be successful, creating "macro-benchmarks" is critical to Benchpress' success.

## Benchmarks for Angular 2.0

Benchpress is designed to answer one question: “Is this code fast enough?” However, wrapped in that question are a number of additional questions which can be answered for individual projects. For Angular 2.0, the additional questions are:

### “How fast is fast enough?”

16.667ms. We want to enable 60fps web apps. Therefore we have 16.667ms to execute any JS code, layout and re-paint.

### “running where?”

Running on modern hardware. We are building for the [near] future. Current “best of class” hardware will be middle-of-the-road in a year’s time. Therefore, we should target the latest mobile and desktop hardware.

### “what code?”

Real code! That is very broad, so we plan on tackling benchmarking goals like a video game: with levels.

## Levels for Angular 2.0

The benchmarks for Angular 2.0 should follow a progression.

### Level 0

It is **possible** to write a smooth, highly interactive app with Angular. The simplest apps (e.g. helloworld) should run at 60 frames per second (see [background on jank](#h.miag1uptgg2x)) out-of-the-box.

* Open question: should this apply to startup as well? If so, we need to be careful about the work that we do in the bootstrap phase. Even parsing the JS may take too long.

### Level 1

The [flagship Angular 2.0 app](https://docs.google.com/document/d/1ofxNVwEpY2xDqpSSmJIyzuaPdWKW2XTYeTPLmkeGG44/edit) should run at 60 frames per second. We are going to write the app, so we control the best practices.

* This would involve creating benchmarks from the app, which is a process that we are working on perfecting.

### Level 2

<suggested by the community> Trying to run at 60fps for all pathological cases should be an explicit non-goal. Instead, we want to take use-cases from interesting apps and large apps (e.g. GreenTea or DFA) and measure ourselves using their code.

### Level n

Make it easy to write any app at 60fps.

* First, we need to understand best practices @60fps.
* And we would need to give app developers tools for incremental rendering.

At this point we don't know all the challenges, but we'll likely need to

* re-engineer ng-repeat to support infinite scroll by default
* ng-if et al: implement block caches so we aren't constructing and destroying the same DOM tree repeatedly
* be much smarter about how we interact with the DOM

## A Mock-up

Ideas are better expressed in code.

Demo: [(demo link on github)](http://jbdeboer.github.io/suite.html)

### Tiny Harness

Real [mobile] hardware is fairly closed. We can’t add a bulky benchmarking harness (e.g. custom builds of Chrome) without effecting results.

Furthermore, meaningful data comes from running on “bare” hardware, as a production app would.

### Real Apps

We want to benchmark code which is a close to production code as possible. Production code doesn’t have a “warm-up” phase. Nor does it run the same code many times in a loop. Benchmarks should not do that either.

### Mock-up Design

To experiment with these ideas, I have constructed a mock-up. There are two pieces to the mock-up: the suite runner and the test runner.

#### Tests

Tests are defined in Javascript, with three parts:

* setup: which configures the DOM and loads any scripts needed for the test
* render: which actually renders the test
* assert: which runs after the test and verifies the page looks like it should

For example: ([context](https://github.com/jbdeboer/jbdeboer.github.io/blob/master/perfs.js#L26))

The “size” concept here generates benchmarks rendering different sized DOMS.

perf('detached', 'Renders a full tree and then attaches it to the DOM')  
 .setup(setup)  
 .render(function(size) {  
 var innerDiv = document.createElement('div');  
 for (var i = 0; i < size; i++) {  
 var s = document.createElement('span');  
 var t = document.createTextNode(i + ' ');  
 s.appendChild(t);  
 innerDiv.appendChild(s);  
 }   
 holder.appendChild(innerDiv);  
 })  
 .assert(function(size) {  
 // Holder should have size children  
 // TODO jasmine integration.  
 if (holder.childNodes[0].childNodes.length != size) {  
 throw new Error('Bad test');  
 }  
 })  
 .usefulSizes(sizesForAll);

#### Suite Runner

([code context](https://github.com/jbdeboer/jbdeboer.github.io/blob/master/suite.html#L75))

The suite runner and the test runner communicate through URL parameters and localStorage. Before the test run, the suite runner loads a list of tests to run into localStorage and then navigates to the first test using window.location.replace().

Since the suite runner navigated to the test runner and each test is a separate page load, no Javascript or DOM state should leak between tests.

While the test is running, there is no monitoring process overseeing the process. This means that the benchmarks are closer to production code. However, there is a risk that we won’t be able to fail cleanly on errors.

#### The test runner

([code contex](https://github.com/jbdeboer/jbdeboer.github.io/blob/master/testrunner.html#L46)t)

1. reads its parameters from the URL and loads the perf test
2. runs the **setup** routine
3. requests an animation frame and runs the **render** routine from the requestAnimationFrame callback
4. requests a second animation frame and measures the time between the first and second frame. If it took 17ms or longer, the second animation frame was pushed out by our code. Therefore our code is too slow.
   1. Measuring the time between the two frame includes the time to run scripts, layout and repaint. This is a critical innovation in Benchpress.
5. after the render routine is finished, run the **assert**
6. writes the test results to localStorage and navigates to the next test, read from localStorage.

When all the tests are complete, the test runner will navigate back to the suite runner which reads the results from localStorage.

#### A Good Benchmark?

When measuring real apps, the setup routine would load the application and configure the state. The render routine would then run a single action; for example an “ng-click” expression from an Angular app.

The benchmark should take the same code path as a user would.

Since it is measuring a single action, it is **actionable**. e.g. “Can we render a pop-up triggered by an ng-click fast enough?”

Since it is following the same code path as a production app, it is **meaningful**.

It is **accessible**. Anybody can navigate to the test suite runner without a special configuration.