Pyjeon Developer Blog

A how-to blog from a couple hackers who embrace Python and JavaScript

Why Pyjamas Isn’t a Good Framework for Web Apps

Posted on July 29, 2012 by Alexander Tsepkov

Earlier this week, I stated that Pyjamas no longer seems like a viable solution for Grafpad (or many other web-apps for that matter). In this post, I will explain the flaws with Pyjamas that ultimately made me decide to switch away from it. I’m aware that Pyjamas project is currently getting an overhaul, and I hope that these flaws get addressed in the upcoming Pyjamas releases. Before I go any further into bashing Pyjamas, I want to mention that I’ve been using Pyjamas for several years, writing over 20,000 lines of Python code that runs inside the browser (as well as several Pyjamas wrappers for extending its functionality). I appreciate the problem Pyjamas is trying to solve, and I definitely think it’s a useful tool. Perhaps one day Pyjamas will be good enough for the browser, unfortuantely it has a lot of issues to solve before that’s the case.

Experienced JavaScript developers might already be familiar with many of the points I will bring up. To summarize Pyjamas’ flaws in one sentence, it basically assumes that JavaScript is still a joke of a language it was several years ago and tries to apply outdated solutions that don’t scale well. Today’s JavaScript, however, can run circles around its predecessor, both in terms of performance and functionality. Many innovative design patterns have also been posted for keeping JavaScript code clean and object-oriented. In some ways, JavaScript has even surpassed Python in terms of design, which still lacks proper private variables, for example. So what are some of the big offenders in Pyjamas?

Browser Detection instead of Feature Detection

Many of you are probably familiar with Pyjamas’ compilation scheme. If not, it basically creates multiple versions of the JavaScript code, one for each major browser (IE, Firefox, Safari/Chrome, Opera) and serves the appropriate one depending on your user-agent string. A quick Google search will reveal thousands of pages explaining the problems with this technique (called browser detection), so there is really no point for me to go into much detail here. The first problem with browser detection is that we assume that the user will be using one of the browsers we’re detecting

(sorry Konqueror). The second problem is that we’re assuming the user is using one of the versions of this browser that still has the same issues/functionality. I’ve already posted about the changes I had to make in Pyjamas to make it use IE9 properly, which has full canvas support, yet Pyjamas still treats it like IE6 (ironically, IE9 actually behaves more like WebKit than IE6). The third problem is that many browsers spoof the user-agent string, pretending to be a different browser (for various reasons). These browsers may support features that the spoofed browser doesn’t support and vice versa, forcing us to use an unnecessary work-around for a feature that the browser supports natively (just like IE9 being forced to use VML instead of canvas), or preventing the feature from working altogether (imagine if Chrome, with no VML support, spoofed IE6 user-agent string).

Bloat and Boilerplate Hell

If you’ve peeked at Grafpad’s JavaScript, you probably saw 80,000 lines of code in a 3.5MB file. But did you know that the pre-compiled version of Grafpad front-end is only about 8,000 lines of code? We have 10 times the needed code just to pretend like we’re still using Python. What’s worse, most of that code is only there to support obscure Python functionality most of us are never going to use in a web-app anyway. Pyjamas has become the most complete Python framework for the browser, unfortunately it has also become the most bloated one, with most other frameworks (such as py2js) only needing to generate 1.5 lines of JavaScript for each line of Python code. You can see the 80/20 principle at work here, where 20% of Python’s features account for 80% of Pyjamas’ boilerplate. In my opinion, it would make a lot more sense to only support the commonly used features of Python, allowing the user to rewrite the bits that don’t work well for JavaScript. After all, the most tedious things to port between languages are the algorithms, not the object structure.

Debugging

In theory, Pyjamas is much easier to debug than JavaScript. Unlike JavaScript, which either throws vague errors or worse yet, silently fails a block of code and continues execution like nothing happened, Pyjamas throws Pythonic exceptions, which most of the time do a very good job pinpointing the exact line that caused the problem… at least when you run your program through Pyjamas Desktop. The problem is, Pyjamas Desktop has been broken for almost 3 years now, requiring you to either use a 3-year old Linux distribution (last known version to have support for python-hulahop) or rely on WebKit or MSXML implementations, neither of which supports canvas.

Alternatively, you can debug your code directly in the browser. Pyjamas sports a good set of Python exceptions emulated in JavaScript through clever use of try/catch blocks. Unfortunately, this alternative not only lacks proper stack trace, but also the original code (and compiling your Pyjamas app with any of the debug modes doesn’t solve this, regardless of what various outdated posts on the mailing list claim). Needless to say, the errors raised by Pyjamas in the browser are not very useful. If you made an IndexError on line 50 of your code by referencing “object.array[5]“, for example, expect Pyjamas to throw some weird error (that’s right, chances are it won’t even realize it’s an IndexError – or at least won’t report it well, the except blocks seem to work correctly) on line 30,000 of your compiled JavaScript, which will reference `$p['getattr'](object, 'array').__getitem__(5)`, among a bunch of other boilerplate which could have caused an error as a result of an earlier error in your code or a Pyjamas bug. Even when debugging using Pyjamas Desktop, the browser errors can occasionally be inconsistent with normal Python (usually due to a bug in Pyjamas), and it’s a pain to

troubleshoot these. And there is really not much Pyjamas can do to remedy this, in my opinion.

Adding additional assertions to catch every possible case to throw Pythonic errors is a fool’s errand no different than trying to parse HTML using regular expressions. Python’s ability to throw relevant assertions stems from its fundamental design. It’s very strict about using non-existing/undefined variables and comparison of irrelevant types. JavaScript, on the other hand, is very lazy/permissive about these, much like Perl. Python is proactive about its assertions, Pyjamas tries to be reactive. It’s unrealistic to forsee every special case that could arise and account for it with an assertion the same way Python would. Even if you manage to do so, you will have added even more code to Pyjamas’ already large chunk of boilerplate (not to mention potential for new Pyjamas bugs). One option is to compile these assertions away when the debug flag isn’t set, but even then you would be doing the exercise of examining all possible errors that Python could throw in each case, plugging in more “reactive” logic to make JavaScript work the same way. Instead, we should make the framework easy to debug in the environment it’s meant to be in. Since we can’t make JavaScript behave like Python, and we can’t do compile-time debugging like we would with C++ or Java, we should make the output easy to understand, so that we can map it back to the original code.

Python is not Java, DOM is not a Desktop

This brings me to my next point. GWT (the original inspiration for Pyjamas) might be more bloated than Pyjamas, but there is something it can do that Pyjamas can’t: compile-time error catching. If it wasn’t for Python being a dynamically-typed language, a lot of my rant in the previous section about debugging would be irrelevant. Additionally, I don’t feel that Pyjamas is approaching the problem from the right angle. Python has the advantage of being much more similar to JavaScript than Java ever will, and a lot of Pyjamas’ wrapper logic wouldn’t even be necessary if Pyjamas didn’t try to pretend to be GWT (in addition to pretending to be Python). GWT was designed to make web development similar to Desktop GUI development, since that’s the background many Java developers come from. What other purpose is there to fake MouseListener and KeyboardListener in an environment that wasn’t designed to need either (KeyboardListener, by the way, is another source of grief for Pyjamas – it’s what makes the keyboard pop-up all over the place on mobile devices, it also attaches a fake input element to the current element, pretending like they’re the same element, adding even more boilerplate and wrappers to the code)? What other purpose is there to build the entire DOM dynamically (which, by the way, is also extremely inefficient)? The browser page was not designed to function the same way your Desktop calculator app does. Anyone who has taken a few minutes to learn how the DOM works probably agrees that it’s actually superior to the old-fashioned Desktop way of writing the GUI. I’m lazy (otherwise I wouldn’t have written my front-end in Python), so when a new technology comes along that clearly makes my life easier, why ignore it?

If it wasn’t for trying to fake a Desktop GUI, Pyjamas wouldn’t need all these wrappers. Most other Python-faking frameworks allow one to invoke JavaScript logic as if it was a regular Python object/function. Pyjamas, on the other hand, requires one to first write a wrapper for Pyjamas Desktop using Python, then for the browser using some limbo version of Python/JavaScript hybrid (where you can’t even access elements of array using standard indexing), and finally rewrite a separate version of your limbo code for each non-compliant browser (definitely IE, and possibly some others). This wrapping might have been necessary in Java, but should not be needed for Python at all,

and could have been prevented with better design. But wait, there IS an alternative! You can put raw JavaScript in your code using JS() method and passing it one giant string of JavaScript code. Unfortunately, that chunk of the code will get completely ignored in Pyjamas Desktop (which you're using to debug your entire app, since the browser debugger is no help at all), and to actually reference anything from this chunk of code in the browser, you will need to reference these variables the same way: "a = JS('a')" (again, don't expect "a" to get set in Pyjamas Desktop). Oh, and don't try to modify any of the DOM elements created by Pyjamas from anything other than Pyjamas, you will run into object state sync issues. Pyjamas wraps each DOM element in a Python object, which then stores the element's state as a set of variables, and assumes it doesn't change without Pyjamas' permission. Pyjamas plays well with other JavaScript frameworks... as long as they don't touch any portion of the DOM Pyjamas uses.

JavaScript has its Strengths

JavaScript might not be the cleanest language, and I still much prefer Python to it. But I must give it credit where credit is due. First of all, it integrates the DOM into itself really well. I can take any DOM element, assign a function to onMouseDown event as if it was a regular JavaScript object, and all of a sudden I got an element that reacts to my mouse clicks. No need for complicated ClickHandlers.

Pyjamas has a lot of abstraction layers, both to hide JavaScript inconsistencies, and make it easier to build widgets. However, native JavaScript libraries, like jQuery, do a much better job at both. Yes, jQuery doesn't scale well for larger projects, but there are libraries that do, like MooTools (which, by the way, was inspired by Python). But realistically, if you create a simple wrapper for generating classes (or loot one from John Resig's blog – the same guy who wrote jQuery), even jQuery becomes good enough for creating large projects. Pyjamas, on the other hand, adds so much abstraction, that sometimes I need hacks just to manipulate the DOM. If you look at the DOM of a typical Pyjamas app, you will notice layers of unnecessary elements: images wrapped in divs, wrapped in more divs, placed inside some table that resides inside yet another table. When I try to render my app on a tablet, it often crashes due to the DOM bloat.

Pyjamas also assumes that JavaScript is slow, which was true when the project first started. As a result, it duplicates parts of its boilerplate code to avoid an extra function call (while adding excessive function calls and abstractions in other places). Ironically, JavaScript engines have come a long way, and a lot of Pyjamas' optimizations are no longer relevant (such as using `object["property"]` instead of `object.property`). In fact, a quick paint app I wrote in Pyjamas actually runs faster in Chrome than Pyjamas Desktop. That same app runs faster still when written in pure JavaScript. It's especially noticeable when using the paint-bucket tool, which works by pixel-scanning and takes a couple seconds in Pyjamas yet almost instantaneous in JavaScript.

Summary

While Pyjamas is the most complete Python emulation in a browser, it has become a very bloated and brittle framework. It doesn't embrace any portion of JavaScript, nor the DOM, trying to hide them away like some sort of deformed beast. By pretending to be pure Python, it not only puts unrealistic expecations on itself, but also fails to make use of good parts of JavaScript. Instead, Pyjamas

embraces a solution designed for a statically-typed language, favoring a GUI structure that should have died a decade ago.

So What’s The Alternative?

I did mention that I am porting Grafpad away from Pyjamas. However, I’m not crazy enough to rewrite the entire project in pure JavaScript. Rewriting all the code in a language with different quirks and troubleshooting differences like division rounding and modulo signage is not my idea of fun. I also still prefer to keep my front-end code interchangeable with the back-end (more or less), which has already provided multiple advantages, such as moving the proprietary clipping and recognition algorithms to the back-end in just a few hours of work. I happen to have another ace up my sleeve. In the next post, I will review multiple alternatives for Pyjamas and explain the solution I’ve chosen.

This entry was posted in **Frameworks**, **Performance** and tagged **Pyjamas** by **Alexander Tsepkov**. Bookmark the **permalink [http://blog.pyjeon.com/?p=302]** .

About Alexander Tsepkov

Founder and CEO of Pyjeon. He started out with C++, but switched to Python as his main programming language due to its clean syntax and productivity. He often uses other languages for his work as well, such as JavaScript, Perl, and RapydScript. His posts tend to cover user experience, design considerations, languages, web development, Linux environment, as well as challenges of running a start-up.
View all posts by Alexander Tsepkov →

3 THOUGHTS ON “WHY PYJAMAS ISN’T A GOOD FRAMEWORK FOR WEB APPS”

Christian Iversen
on **September 3, 2012 at 10:21 am** said:

Hi Alexander

I’m Christian Iversen, and I’m the leader of a crack team of developers working on exactly what you’re looking for – a light-weight alternative to Pyjamas.

We’ve create Pyjaco (The PYthon to JAvascript COmpiler), and we’d be very happy if you could test it alongside the other alternatives you’ve found.

Our reasons for creating Pyjaco instead of just improving Pyjamas are almost identical to the points you mention here. I should mention that Pyjaco originates from Py2JS, so it shares some of the strengths (low LoC overhead), but it shares almost none of its weaknesses, and is essentially a rewrite.

The current version (git-HEAD from here: https://github.com/chrivers/pyjaco) should work fine for your needs, but we are working on Pyjaco 2.0 (internally known as the “IST version”) which greatly enhances the quality of the generated code, while keeping almost the entire stdlib implementation (so it’s not a reinvention of the wheel).

If you need any help at all getting started, I hope you will contact us at pyjaco@googlegroups.com.

Alexander Tsepkov
on **September 17, 2012 at 12:10 pm** said:

Hey Christian,

Thanks for the response. If I got my facts correct, I think Pyjaco is a fork off of py2js (variation 3) I mention in my next post? I’m guessing the legal issues have been resolved then (or the code in question has been rewritten)? Let me know if you want me to update that section or add a separate one for Pyjaco. As I mentioned, I am quite impressed with its functionality, my main pet peeve is needing stubs at compile time for external libraries.

Christian Iversen
on **September 28, 2012 at 7:37 am** said:

Hey, sorry! I didn’t see your reply before now.

Pyjaco is almost a complete rewrite of py2js (variation 2). Many parts of the stdlib are written 100% from scratch, and most of the compiler, too. There are certainly no legal issues using Pyjaco, although we are seeking to rewrite the last parts that are not 100% hours, so that we have full control over the source code and license.

And yeah, it’d be great if you could update the name to Pyjaco. Also, we have a google group, and a homepage: pyjaco.org. It’s minimalistic, but has a live demo!

When you say stubs, you are referring to JS() and @JavaScript? That’s

true, that’s the biggest problem with the current compiler.

However, we have a new version coming up, the IST compiler, which will solve this. In this next version, you can simply refer to anything as js(x), and use js semantics in that subtree.

For example:

pythonic:
document.foo() -> document.PY$**getattr**(‘foo’).PY$**call**()

javascriptish:
js(document).foo() -> document.foo()

This way, it will be very easy to integrate into any existing libraries, without needing any kind of “stubs” or “js header files”.

I hope this answers your question?