JavaScript isn't a bad guy

A little over a week ago I read an interesting article that appeared on Hacker News about how the Googlebot crawls JavaScript, which was quite informative and interesting although probably only to those who write JavaScript powered websites/applications on a day-to-day basis - like me.

What I found even more interesting were some of the reactions on the Hacker News comment thread. I was quite surprised (well, I am quite naive) to see some of the discussion contain some anti-JavaScript sentiment. This is probably a good point to say that I am very much pro-JavaScript and consider it a technology that has driven web progress from static documents to interactive experiences (like google maps).

I'm going to pick on someone now.

> IMHO: Websites that don't have "realtime" content should always stick with traditional HTML. I'm a Webdeveloper myself and i don't like the JavaScript Frontend trend.
>
> Many Devs use Frontend JS in places where it's absolutely not needed. If you're building an App that updates in realtime, shows informations while it's created, i'm fine with Frontend JS, but it's an overkill for most content pages.

axx

When I read this it sort of blew my mind. How can a developer be against client-side JavaScript for anything other than realtime content?! Clearly we disagree.

I would go as far to say that JavaScript can be used to enhance almost any interaction that changes the state of a "widget". That goes for presenting more information, facilitating CRUD actions, and creating rich media experiences such as visualisations. *It's just we've only recently started to get good at doing it quickly*.

Crafting effective experiences with JavaScript

Designing how we consume and interact with the web fundamentally changed when AJAX became a thing. Click a button and watch something happen onscreen. Then iPhone apps happened and they were doing it too... and better?! Ever since, the web has looked slow by comparison and everyone points fingers at JavaScript for being rubbish.

It's a bit unfair though! We definitely can write performant JavaScript, it's just historically we haven't. We could pontificate forever why that was, but I'd rather put a few easy to implement ideas out there and get some discussion going!

Data first approach

A lot of problematic JavaScript I've encountered in production can be attributed to data bloat. The more data you need, the more data you have to wait for - that bloat counts on page load, per frame, and on clean up. There's a few common things that come up when looking to optimise data and most can be dealt with server side:

1. GZIP data being sent
2. Remove unused properties from APIs where possible.
3. Move non-unique properties in repeated objects to a 'summary' type object.
4. If data is available on the server (and you know the client will need it immediately) send it inline with the page.
5. Defer loading of data not exposed to the user immediately to a subsequent API where possible.

Inline data

Here's a todo list component.

[Image of todo list]

Back in them olden days, we might have had the server render the markup and then use JavaScript to extend it. But, it's dynamic enough that reading/writing from the DOM isn't trivial. The features of this todo list include:

- Creating items
- Deleting items

- Editing items
- Reordering items

The benefit of rendering the markup on the server that we don't want to lose is the data being on the page as soon as it loads. A lot of applications make the mistake of serving an empty page very quickly and then fetching all the data after the initial load. An alternative is to inline the data like so:

```
<script>
    var todoItems = [
        {
            id: '14a29154-4be5-45e7-aa65-40585f94cd85'
            value: 'Make Espresso',
            done: true
        },
        {
            id: 'da8ee3c9-1701-4547-8dfe-b6c4ee4f4893'
            value: '2x20 intervals',
            done: false
        }
    ];
</script>
```

While we can't display that data as soon as the page loads, we can display it as soon as the JavaScript executes, which is pretty quick and can be optimised for.

Deferred Data

Let's add a feature to the todo example above which allows people to add notes to a todo item. We could just extend our todoItems array that's in the page:

```
<script>
    var todoItems = [
        {
            id: '14a29154-4be5-45e7-aa65-40585f94cd85'
```

```
            value: ‘Make Espresso’,
            done: true
        },
        {
            id: ‘da8ee3c9-1701-4547-8dfe-b6c4ee4f4893’
            value: ‘2x20 intervals’,
            done: false,
            note: ‘Meeting Brad at cemetery junction
9:30’
        }
    ];
</script>
```

But the todo list UI doesn’t allow users to see this detail until the interact with the specific todo item. Not adding this property means less data to query for, less data to send, and less data to load into memory.

The alternative is to create an API for getting further information or ‘detail’ on a particular todo item - something like this:

```
GET /api/todo/details/:id

{
    note: ‘Meeting Brad at cemetery junction 9:30’
}
```

Obviously if this API is slow then the user experience will suffer. The data is pretty small, so it’s TTFB (time to first byte) we should monitor and optimise for on the server.

Thinking in components

The JavaScript in legacy websites and applications I encounter often looks pretty similar:

1. There's a lot of it
2. It's grouped by "feature" if grouped at all
3. There's a lot of duplicate/similar code
4. The vast majority of SLOC is focused around reading/writing from/to the DOM
5. It generally suffers from a lack of architecture and planned development

It's not like all the previous maintainers of these websites were bad either - they're probably representative of the average front-end developer at the time. It's pretty obvious that frameworks can help solve most of these issues but people still aren't using them even when they have a fair amount of JavaScript to maintain.

I think one of the reasons people shy away from frameworks is because they find they restrict the speed they can roll out code at the beginning of a project. You do have to architect the application you're going to write if you want to create a good codebase. But given the issues above, I think this is a beneficial process.

JavaScript frameworks - well the good ones anyway - promote development in terms of components rather than features. When you write a component to do one job and it does it well, it can be used in multiple places. I like to think of this as an application of the DRY principle, although I'm not sure how accurate this is.

There's something to be said for the added intricacy of component oriented architecture. Components will likely consist of many sub-types working together - like Angular's directives and services. These tools must be used optimally lest we end up with an application endlessly sprawling outwards with no code re-use. Components allow for easy code reuse, but they also make it easier than ever to just add another module without causing regression elsewhere.

Done right, a framework can solve issues 2 - 5 and maybe even issue #1 if your framework coaches you to write expressive code; it's no guarantee however.

The DOM is a poor data storage tool

It seems like we've used the DOM to store information and JavaScript to manipulate the DOM forever. It made sense when JavaScript adoption was low,

search engines only indexed source HTML, and there were many accessibility concerns - but none of these are true anymore! There are some clear disadvantages to relying on the DOM for data storage.

1. You have to write JavaScript every time you want to read/write from the DOM. That's a lot of JavaScript.
2. Changes to the DOM have knock-on effects including repainting - the most costly aspect of runtime performance.
3. Some libraries *cough* jQuery *cough* are a bit opaque and make it easy to consecutively write/read to/from the DOM without the developer realising.
4. There may be many pieces of JavaScript modifying the DOM, so data you read has to be validated as well.
5. It's very difficult to craft consistent behaviour around inconsistent data. It's easy to introduce bugs into DOM driven applications.

Recent frameworks have turned to templating for a solution which provides flexible data binding for DOM behaviours. For most this is easier to maintain and perhaps even to write performant code for.

jQuery vs Angular

When we bind events to elements in jQuery we're working with the target element unless we traverse the DOM for a wider scope, or bind the click to a parent element and query the DOM for child elements. This means we have to attach the data to the object we're binding to, like so:

```
<button class="delete" data-id="da8ee3c9-1701-4547-8dfe-
b6c4ee4f4893" type="button">
    Delete
</button>

    // Bind this function to every instance of '.delete'
on this page
$('.delete').on('click', function(e) {
            // Extract ID from target
    var todoId = $(e.target).data('id');
```

```
    // Fn() that sends AJAX request to delete todoID
    deleteTodo(todoId);
});
```

With Angular we can work with repeated elements that have a wider scope and can pass that scope around, no querying necessary.

```
<div class="todo-item" data-ng-repeat="todo in
todoItems">
    <button class="delete" data-ng-click="delete(todo)"
type="button">
        Delete
    </button>
</div>
```

There are some obvious benefits to this approach.

- There is less JavaScript to write/maintain because the framework handles the DOM mutation.
- You get a clear understanding of how the component is glued together by just viewing the source HTML.
- This follows DRY principles where we reference one object the application already knows about, rather than using new data read from the DOM.
- It's generally a much more expressive piece of code.

A sense of maturity

In a lot of ways it feels like this turn towards templating, frameworks that enforce MVC or MVVM approaches - or *heck* just enforcing a pattern at all - is a sign that Front-end development is growing up.

While we can still throw script on a page and have it work, we have the option to use tools that help craft better code and experiences - though a little discipline is required.